INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

NOVEL CACHE REPLACEMENT ALGORITHMS FOR HIGH PERFORMANCE COMPUTER SYSTEMS

BY

HUMAYUN KHALID

A dissertation submitted to the Graduate Faculty in Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1996

This manuscript has been read and accepted for the Graduate Faculty in Engineering in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

6-18-1996 — Date

M. S. Obaidat — Professor Mohammad S. Obaidat, Chair of Examining Committee

6/18/96 — Date

Gerard G. Lowen — Professor Gerard G. Lowen, Executive Officer

Professor Richard Tolimieri

Professor Michael Conner

Professor Roger Dorsinville

Professor Mokhtar Boukli-Hacene

Dr. Mohammad T. Fatehi

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

NOVEL CACHE REPLACEMENT ALGORITHMS FOR HIGH PERFORMANCE COMPUTER SYSTEMS

by

Humayun Khalid

Adviser: Professor Mohammad S. Obaidat

This thesis presents novel cache memory replacement algorithms for high performance computer systems. The proposed algorithms use neural networks (NNs) for the purpose of guiding the line replacement decisions in caches. Key to our algorithms is the correct identification and displacement of inactive lines. This allows our strategy to provide better cache performance as compared to the conventional LRU (Least Recently Used), MRU (Most Recently Used), and FIFO (First In First Out) replacement policies.

Performance of the neural network based algorithms was tested against the performance of existing popular

strategies through extensive trace-driven simulations in order to obtain reliable results for a wide spectrum of practical cache configurations. Simulation results show that the proposed algorithms provide substantial performance improvement in the miss ratio over the conventional algorithms for several benchmark programs. Excellent performance of neural network based replacement strategies means that the new approach can be adopted as future cache replacement algorithms. Good results, provided by the line replacement algorithms, indicate that our schemes have potential for providing promising results when applied to the page replacement algorithms in virtual memory systems and disk caches.

For the sake of Allah,

The Beneficent, The Merciful

ACKNOWLEDGEMENTS

I would like to thank and gratefully acknowledge my adviser Professor Mohammad S. Obaidat. His patience, guidance, and vast knowledge were of enormous help in achieving the research goals.

Sincere thanks are due to Professors Richard Tolimieri, Michael Conner, Roger Dorsinville, Mokhtar Boukli-Hacene, and Dr. Mohammad T. Fatehi for serving on my doctoral committee. Many thanks go to my colleagues at the Computer Engineering Research Laboratory for their friendly support and valuable assistance.

Last but not least I would like to thank and pay my respect and love to all my family members for their endless support and enthusiastic encouragement. My special appreciation is due to Ms. Khadeja Zamir, Dr. Aftab Ahmed, Mr. Murad Khalid, and Ms. Arjumand Khalid.

TABLE OF CONTENTS

List of Figures

CHAPTER 1

NOVEL CACHE REPLACEMENT ALGORITHMS FOR HIGH PERFORMANCE COMPUTER SYSTEMS

1.1 Introduction

A cache is a small, costly, high speed buffer, logically placed between the processor and main memory. It is used to hold data and instructions from actively referenced sections of main memory. Due to the size differential between main memory and the cache, replacement algorithms exist in order to manage copies of those portions of main memory that reside in the cache. As the contents from sections of main memory are brought into the cache, the replacement algorithm must decide the contents in cache that need to be discarded to make room for the newly arrived items. In most of the literature, these mobile items comprising of several bytes of data and instructions are referred to as lines/block frames.

Traditionally, most of the practical caches have incorporated one of the following line replacement

policies: LRU, MRU, and FIFO. The most popular among the aforementioned strategies is the LRU replacement scheme. As new lines enter the cache, LRU algorithm discards the cache line that was referenced furthest in the past with the hope that the discarded line is not going to be referenced in the near future. The performance of LRU algorithm is, however, not the ultimate. Our research work shows that for the present day benchmark programs, representing typical computer workloads, it is possible to develop practical algorithms that can provide performance improvement over the ubiquitous LRU scheme. The upper limit on the performance of such practical replacement algorithms is set by the optimal algorithm (OPT). OPT algorithm was first described by Belady et al. [1,2] and further studied by Mattson et al. [3]. The OPT replacement algorithm is a lookahead scheme since it always discards a cache line that is referenced furthest into the future. Although, OPT is a non-realizable scheme because of the fact that it has perfect knowledge of the future line reference behavior yet, it provides a metric for evaluating and investigating the performance of other practical strategies. Understanding the relative behavior of different cache replacement schemes in an effort to develop new and improved

strategies is the subject of this thesis.

1.2 Statement of the Problem and Contribution of this Thesis

This thesis focuses on the line replacement problem present in fully associative and set associative caches. In particular, we have concentrated on alleviating the problem in set associative caches which exist in most of the high performance machines. We have selected set associative caches because of their practicality and wide use due to the good cost-performance ratio provided by such caches. The schemes presented in this thesis can be easily tailored for use with fully associative caches. Infact, fully associative caches can be regarded as set associative caches having set size equal to unity.

The line replacement problem is a critical issue in high speed computer architectures. It asks the following question: which line should be replaced when a cache miss occurs and the cache set, to which a miss occurs, is full ? There are several line replacement schemes,

mentioned earlier, that suggest different strategies for coping with the problem efficiently. Generally, for most cache sizes and computer programs, LRU performs better than all the other existing policies. However, LRU is known to perform poorly for programs exhibiting more of Spatial Locality than Temporal Locality. Due to the low performance of LRU, cache sizes have increased over the years in order to get better values for cache miss ratios. Unfortunately, memories become slower as they get larger. This has resulted in increased cache cycle time and is responsible for offsetting the improved cache performance obtained through lower miss ratios. Also, the VLSI technology imposes constraints on the amount of circuitry that can be put on a given chip area. This means that the size of a cache that can be accommodated on a chip containing the central processing unit (CPU) is restricted to small and moderate levels. To alleviate the problems arising form large on-chip caches, researchers have developed multi level caches. Multi level caches have been successful in solving to some extent a few of the problems stated above but, they have created several other problems such as: management of mutli level caches, maintenance of data coherency between different caches, etc.

Our primary contribution in this research is to develop line replacement algorithms for single level caches that exhibit near optimal behavior for a wide range of cache sizes. This will result in cache performances that would be suitable for workloads requiring high speed computer architectures, thereby, precluding the need for multi level caches. A few other significant contributions are:
(1) The introduction of neural networks to solving a problem in computer architecture, (2) Reformulating the line replacement problem in caches as the one related to the recognition of live and dead line, and (3) Development of a generalized version of neural network based line replacement algorithm suitable for both the estimating and the non-estimating classes of neural networks.

1.3 The Approach

The goal of this thesis is to construct new cache replacement algorithms that are superior to LRU by understanding the intrinsic differences and commonalities between LRU and OPT replacement policies, and using the Temporal and Spatial Locality of reference

characteristics associated with almost all the computer programs. To understand and analyze the relative behavior of LRU and OPT, we performed simulation experiments by modeling two different types of caches simultaneously, one using LRU and the other using OPT as the replacement algorithm. This comparative modeling allowed us to observe instantaneous changes between the two caches in order to obtain insights regarding the approximation of OPT. Analysis of the results from simulation runs helped us to reformulate the line replacement problem in caches. Due to the reformulation of the problem, we were able to use neural networks. Two new neural network based line replacement algorithms were developed that are suitable for two different classes of neural networks (NNs). The third algorithm was a derivative of LRU, and it was found to be appropriate for all classes of NNs. Simulation techniques involving trace files from real computer workloads were used extensively throughout this thesis for reliable results.

1.4 Background and the Literature Review

Computer systems are, in general, made up of three fundamental units: Central Processing Unit (CPU), Memory, and an Input/Output Unit (I/O) [figure 1.1]. A computer program is a set of instructions for a computer system to perform the desired task. The process of executing a program involves repeatedly fetching instructions from the memory and bringing it into the CPU, retrieving operands specified by an instruction, performing the desired operation on the operands, and finally storing the results. The execution time for a program is thus critically dependent on the speed with which memory can be accessed.

$$\left(\begin{array}{c}\text{Execution time}\\ \text{for a program}\end{array}\right) = \left(\begin{array}{c}\text{Instruction}\\ \text{count}\end{array}\right) \times \left(\begin{array}{c}\text{Cycles per}\\ \text{instruction}\end{array}\right) \times \left(\begin{array}{c}\textit{cycle}\\ \textit{time}\end{array}\right) \qquad (1.1)$$

$$\left(\begin{array}{c}\text{Cycles}\\ \text{per instr.}\end{array}\right) = \left(\begin{array}{c}\text{CPU}\\ \text{cycles per instr.}\end{array}\right) + \left(\left(\begin{array}{c}\text{References}\\ \text{per instr.}\end{array}\right) \times \left(\begin{array}{c}\textit{Cycles}\\ \textit{per reference}\end{array}\right)\right) \qquad (1.2)$$

In high performance computer systems, bandwidth of memory is often a bottleneck (the Von Neumann Bottleneck) because, it plays a pivotal role in affecting the peak throughput. This is due to the fact that the memory is much slower than the CPU since its goal is to accommodate large sets of data and instructions. Therefore, the technology used for constructing memory is generally slow, cheap, and dense.

The mismatch between the speeds of CPU and memory is alleviated by efficient hierarchical memory systems [4]-[14]. These systems are composed of a mix of memory devices that range in cost and performance [figure 1.2]. The aim of a well designed memory system is to perform as if it were made up of fastest devices in the hierarchy, yet, its cost should be dictated by slower, less expensive units [see Appendix A].

Cache is the simplest cost-effective way to achieve high speed memory hierarchy and its performance is extremely vital for high speed computers [15]-[18]. Caches are small fast memories that comprise the first level in a hierarchical memory system in most of the present day machines. It is for this reason, that caches

have been extensively studied since their introduction by IBM in the system 360 Model 85 [19]. Caches provide, with high probability, instructions and data needed by the CPU at a rate that is close to the central processing unit's (CPU's) demand rate. They work, because, programs exhibit a property called locality of reference [20]. According to this principle: Information that would be used in the near future will most probably consist of the information in current use (Temporal Locality), and the information adjacent to that in current use (Spatial Locality). The first type of locality property, called Temporal Locality, is due to the program loops in which both instructions and data are recycled. The second type of locality, called Spatial Locality, is due to sequential property of programs, that is, related data items are usually stored together and instructions for a given task/sub-task are generally executed in a sequence.

Proper design and implementation is very important for effective use of caches. Cache design is evaluated via few popular yardsticks such as: Hit Ratio, Miss Ratio, and Cache Access Time. Most of the current literature use hit/miss ratio as the primary indicator of cache

performance [21]. Therefore, we will also evaluate cache performance on the basis of hit/miss ratio. Hit and miss ratios convey the same information from different perspectives. Hit ratio gives the probability of finding the referenced information in cache, whereas, the miss ratio provides us with the probability of not finding the referenced information in cache. Therefore,

hit ratio = 1 - miss ratio ..... (1.3)

A crude expression that provides the relationship between effective memory hierarchy cycle time/access time and hit ratios is as follows:

$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i + \Delta \quad \text{..... (1.4)}$$

$$= f_1 t_1 + f_2 t_2 + \text{.....} + f_n t_n + \Delta$$

$$= h_1 t_1 + (1-h_1) h_2 t_2 + (1-h_1)(1-h_2) h_3 t_3 + \text{...} +$$

$$(1-h_1)(1-h_2)\text{...}(1-h_{n-1}) h_n t_n + \Delta \quad \text{..... (1.5)}$$

Where,

T_{eff} = effective memory hierarchy cycle time.

f_i = access frequency for the i^{th} level of memory.

h_i = hit ratio at the i^{th} level.

$m_i = 1-h_i$ = miss ratio at the i^{th} level.

Δ = cycle time due to secondary factors.

n = total number of levels of a memory hierarchy.

Secondary factors are defined as the factors for which small changes in the values are insignificant to the system. Some of these factors are as follows:

(1) Time delay due to leading and trailing edge effects.
(2) Queueing delays to main memory in a multiprocessor system.
(3) Time to maintain cache consistency in virtual address caches.
(4) Time to satisfy a write hit or a write miss.
(5) Cache cycle time delay in order to maintain multiprocessor memory consistency.

It may be noted that in equation (1.5), the following two relations hold:

(1) $h_1 < h_2 < h_3 < ..... < h_n$ are independent random variables with values between 0 and 1.

(2) $f_1 >> f_2 >> f_3 >> ..... >> f_n$, i.e. inner levels of memory are accessed more often than the outer levels.

In general, we may classify caches as being of uniprocessor or multiprocessor type. Uniprocessor caches may be classified in several ways depending upon several cache features [22]:

1. Unified Cache vs. Split Cache
   (examples: Instruction and Data caches)

2. Tag and Index type
   (examples: R/R, R/V, V/R and, V/V. Where, R means real and V means virtual [23])

3. Through Feature vs. No Through Feature
   (examples: store through vs. no store through and load through vs. no load through)
4. Write Update vs. Write Purge
5. Write Allocate vs. No Write Allocate
6. Single Level vs. Multi Level

Three basic cache organizations were identified by C.J. Conti [24]: direct mapped, fully associative, and set associative. These organizations determine the placement policy/mapping function for blocks of information that map from main memory to a cache

location. To differentiate cache blocks from main memory blocks, we will use the term line/block frame for blocks in a cache. The simplest of all organizations is direct mapping. In this scheme block j of the main memory (B_j) maps to line i modulo N (L_i) of the cache. Where, N is the number of sets in a cache.

$$B_j \text{ ----> } L_i \quad \forall \quad i=j \text{ modulo } N$$

Thus, the mapping of main memory blocks is unique for a direct mapped cache. This scheme is very rigid and easy to implement, however, its performance is generally found to be low. Memory address in this scheme, consisting of a total of s+w bits, is divided into three fields: tag, block, and word fields. Lower order w bits (representing word field) of the address specify the word offset within each block. The upper s bits of a memory address is partitioned into middle order and higher order bits for the identification of tags and the blocks. The middle order r bits (representing block field) are used to implement the modulo N placement, where $N=2^r$. The remaining most significant s-r bits (representing tag field) are used to preserve the mapping between main memory and the cache. This is needed, because, several main memory blocks may map to same cache line. Therefore, to determine whether the

required reference is present in a cache, we have to compare tag field (s-r bits) of the incoming reference with the tag field of a line to which the reference would map in a cache. A cache hit occurs when the tags match. In the case of cache miss, a new memory block is brought into the cache to replace the old block frame. Figure 1.3 shows memory address partitioning and block mapping for a direct mapped cache. This type of cache organization has been implemented in several machines such as : VAX /880, IBM System /370 Model 158, and first level I-Cache (instruction cache) of DEC's Alpha chip.

Another extreme kind of cache organization is the fully associative cache organization. Performance of a fully associative cache is the best. However, it is the most expensive organization. In a fully associative cache any block in memory (B_j) can be mapped to any line in cache (L_i);

$$B_j \text{ -----> } L_i \quad \forall \ (i,j)$$

Memory address is divided into two fields: tag field (s bits) and the word field (w bits). Tag fields for all the lines in a fully associative cache are compared with the tag field of the referenced item on each memory request to determine cache hit or miss situation. The

comparison is time consuming if the tags are compared sequentially. An alternative approach, widely used, is to perform parallel comparison with CAMs (Content Addressable Memories). Due to the use of CAMs in the implementation of fully associative caches, the cost of such caches are high. Performance is achieved through the use of a suitable line replacement strategy. Several practical replacement strategies such as: LRU (Least Recently Used), FIFO (First In First Out), MRU (Most Recently Used), FINUFO (First In Not Used First Out), and Random have been developed by researchers to get a high hit ratio for such caches. Thus, to achieve a reasonable cost-performance ratio for a fully associative cache, we need to restrict its size to moderate levels. Figure 1.4 depicts memory address partitioning and block mapping for a fully associative cache. This type of cache organization has been implemented in several machines such as : MIPS R2000 and Zilog's Z-80000.

A compromise between the two extreme cache organizations is called set associative cache organization. Several commercial high performance machines such as : Honeywell 66/60, Amdahl 470/V7, IBM

370/168, IBM 3033, DEC VAX 11/780, Amdahl 470/V6, and Intel i860 D-Cache (data cache), use set associative caches. A design alternative to set associative cache is the sector mapping cache. In this cache organization, memory address (s+w bits) is again partitioned into three fields: tag field (s-d bits), set field (d bits), and the word field (w bits) as shown in figure 1.5. Figure 1.5 also illustrates that a block B_j in main memory can be mapped to any one of the lines L_i within a set S_k of size A (associativity of a cache) lines. For a cache with M sets, we can write the mapping strategy for set associative caches as follows:

B_j ------> L_i, $L_i \in S_k$ for k=j modulo M

On a cache miss, any one of the line replacement algorithms can be used to displace one of the occupied lines within the corresponding completely filled set. This leads to the following benefits that a set associative cache provides over the two previous approaches:

1. A-way set associative cache is a relatively economical organization as compared to fully associative cache since, the search is only limited to A lines within a set S_k.

2. The implementation is relatively easy.

3. Replacement schemes can be used, within a set, to displace relatively inactive lines as opposed to the rigid direct mapped case. This may result in a higher performance for set associative cache over the first approach.

4. It allows TLB (translation lookaside buffer) operations to be overlapped with the searching operation within a set (see figure 1.6).

The choice of a cache organization can have significant impact on cache performance and costs. In general, set associative cache organization offers a good balance between hit ratios and the implementation cost. Also, the selection of a line/block frame replacement algorithm, in set associative caches, can have a significant impact on the overall system performance [21]. The common replacement algorithms used with such caches are : FIFO, MRU, Random, and the LRU. Another replacement algorithm is OPT (optimal) algorithm. It is a lookahead algorithm, non-realizable, that provides a benchmark on which the relative

performance of the aforementioned realizable algorithms is measured. Many different approaches have been suggested by researchers to improve the performance of replacement algorithms [25]-[27]. One such proposal was given by Pomerene et al. [25]. Pomerene suggested the use of a shadow directory when making decisions with LRU. The problem with this approach is that the size of the shadow directory limits the length of the history consulted. Chen et al. [26] studied the improvement in cache performance due to instruction reordering for a variety of benchmark programs. The instruction reordering increases spatial and sequential locality of instruction caches and, thereby, results in a slightly better performance of the replacement and mapping algorithms. Hiremath et al. [27] proposed a set of Fuzzy replacement algorithms (FRA and MFRA) for cache memories. Their algorithms were based on the application of Fuzzy logic technology to the replacement of cache lines. Performance of the Fuzzy replacement technique was evaluated to be comparable with that of a popular traditional LRU scheme, yet, their approach opens a new direction for further research in managing scarce resources such as: disk caching and page replacements in virtual memory systems.

Although extensive simulation, analytical, and experimental work has been done aiming to get a better understanding of the process involved in achieving high performance cache [1]-[193], little is known about the line replacement process. There are many questions that still needs to be answered; a few of which are as follows:

(1) For the present day workloads, is it possible to develop algorithms that have higher performance than the ubiquitous LRU algorithm used in a variety of state-of-the-art machines ?

(2) Are there a few interesting ways in which we can look at the problem of line replacements in cache ?

(3) Is there a tool that can be used to solve our problem ?

(4) What classes of tools are suitable for the problem?

(5) Can we develop algorithms based on all the classes of tools ?

(6) What is the relative improvement that our algorithm(s) can provide over the conventional algorithms ?

(7) Can we develop algorithms that could complement the decision of LRU replacement policy ?

(8) Is there a way we can look at practicality of our

constructed algorithms ?

(9) Does the new set of algorithms show stable behavior for a variety of benchmark programs ?

(10) What is the performance enhancement (if any) the devised algorithm, based on LRU, have over the traditional algorithms ?

This thesis is an attempt to find answers to the questions posed above.

1.5 Overview of the Thesis

We began this thesis by describing the problem. Contributions by the thesis and the literature review was also presented in chapter 1. The next chapter starts by providing a brief introduction on a few popular conventional line replacement algorithms. Line replacement problem is studied and analyzed for the LRU and OPT algorithms using the simulation techniques. Knowledge gained from the results provided by trace-driven simulations is then used to reformulate and understand the problem from a different perspective. Lastly, in chapter 2, the problem is reformulated so

that it becomes suitable for neural networks. In chapter 3, we will begin with an introduction to neural networks. Neural network based cache replacement algorithms are presented for non-estimating and estimating classes of neural networks. Chapter 3 concludes by glancing at the implementation aspect of the proposed algorithms. Chapter 4 basically deals with the performance studies of the conventional, and the proposed non-estimating and estimating neural network based replacement algorithms (NEA and EA algorithms).

In chapter 5, a generalized neural network based algorithm, GA algorithm, is proposed. The GA algorithm is a derivative of the LRU scheme, and is suitable for all the stated classes of NNs. Performance of the GA strategy is tested for two types of trace files: (1) TPC.DATA and BI.DATA trace files which are used for evaluating the performance of the NEA, EA, and the conventional algorithms, and (2) Trace files from the Standard Performance Evaluation Corporation (SPEC). Also, confidence on the simulation results is evaluated. Lastly, chapter 6 presents conclusions and recommendations for future research.

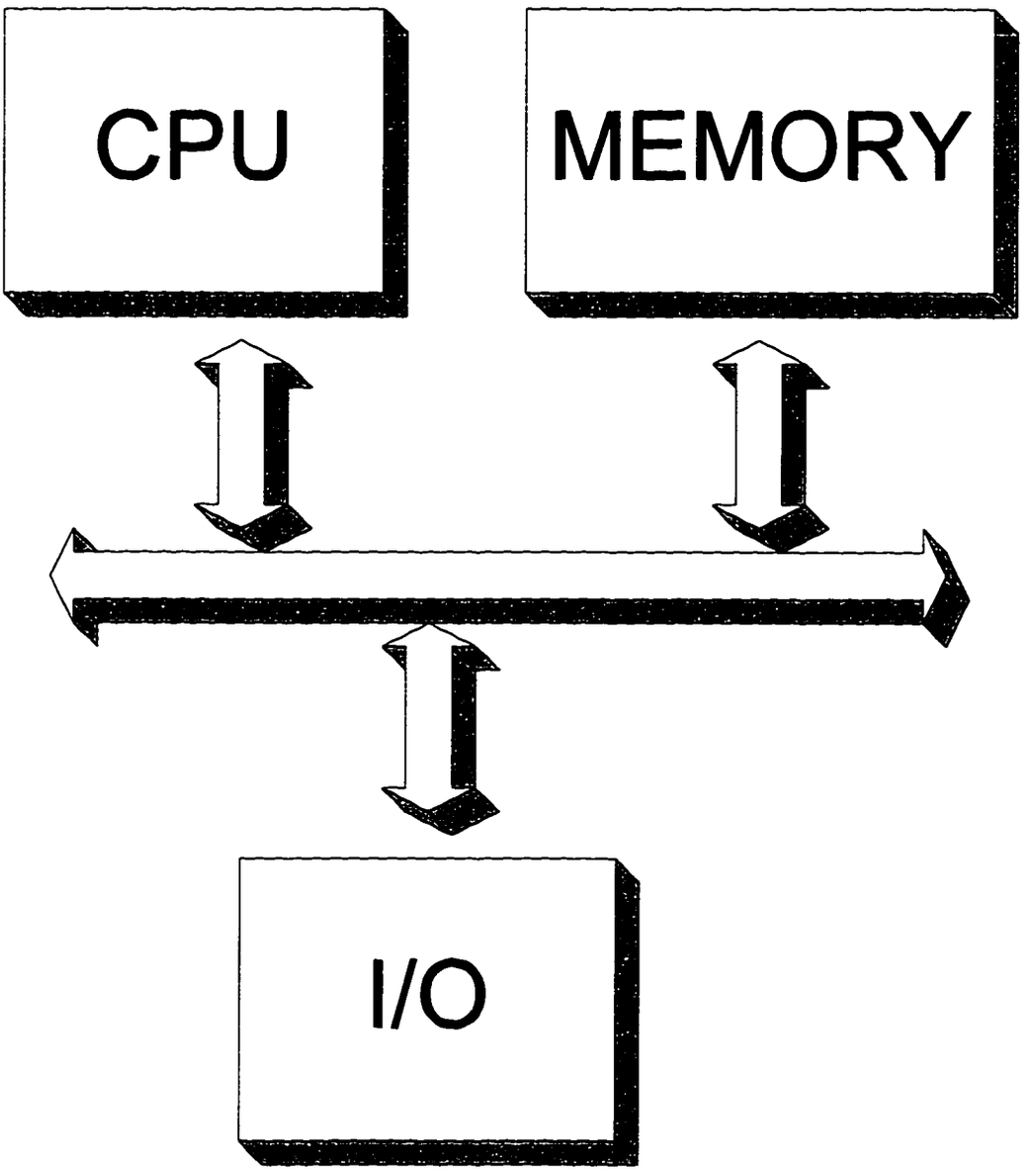


FIGURE 1.1: A TYPICAL COMPUTER ORGANIZATION

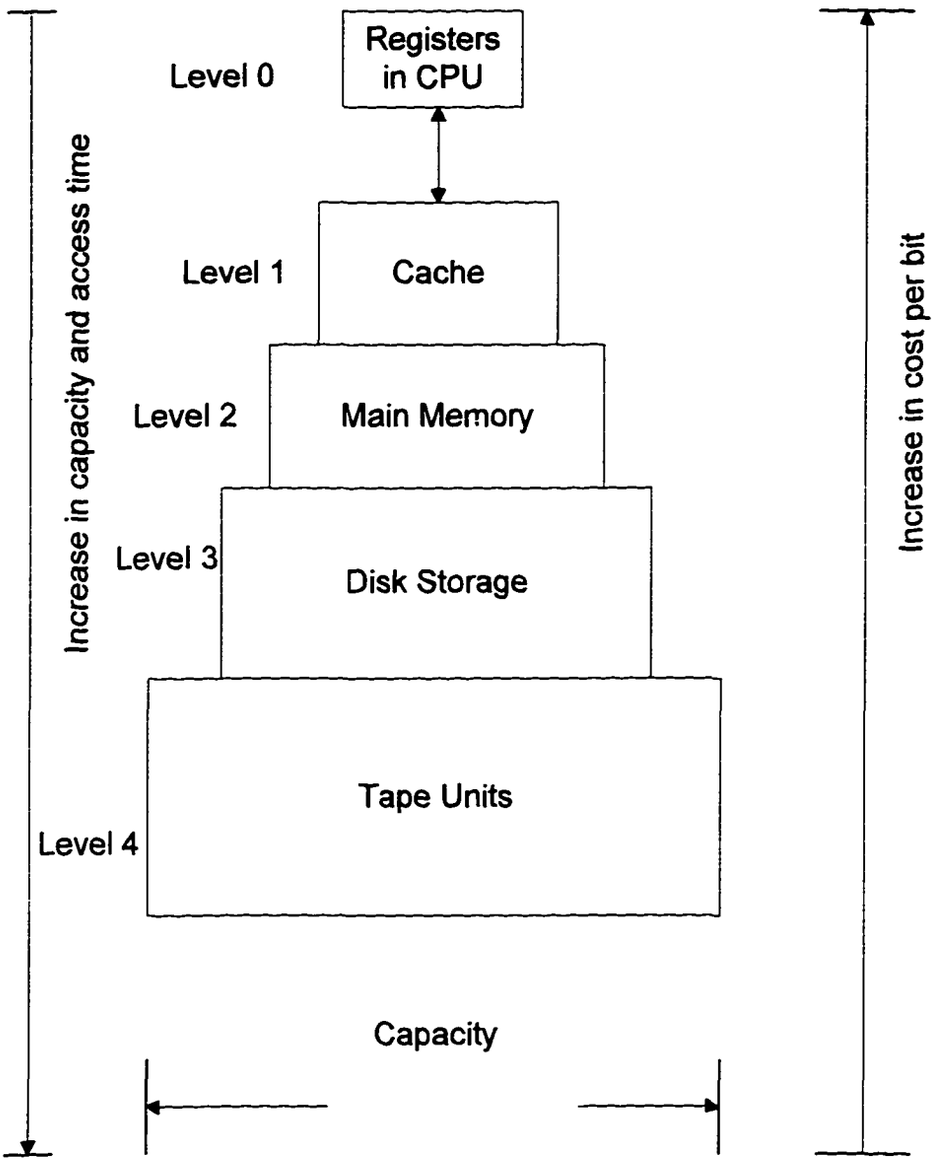
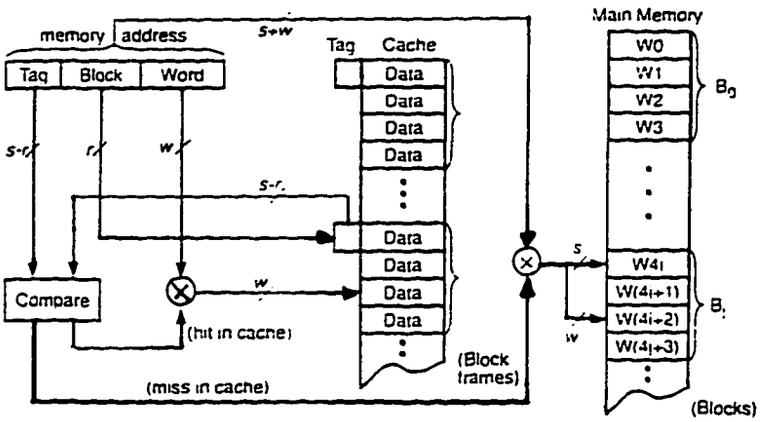
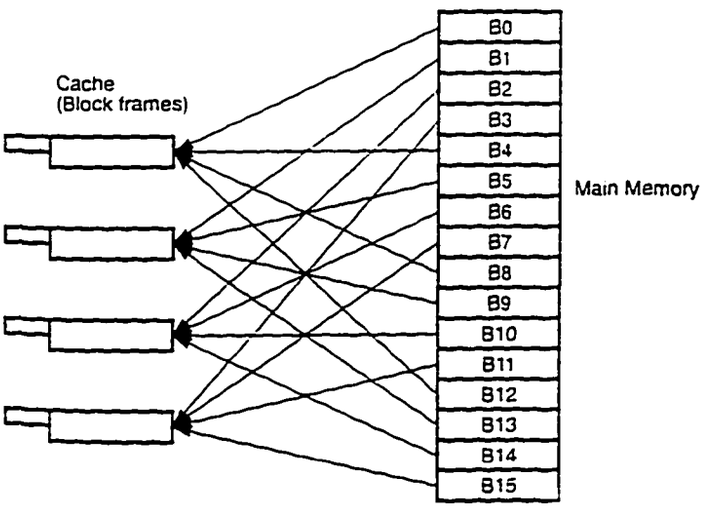


FIGURE 1.2: A TYPICAL FOUR-LEVEL MEMORY HIERARCHY

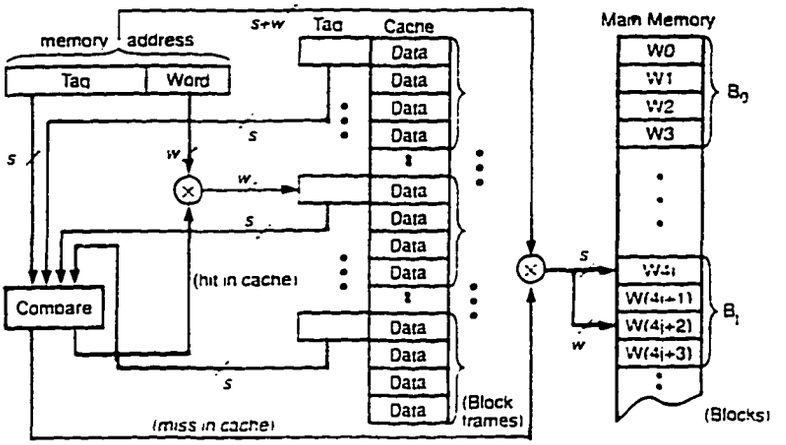


(a) The cache/memory addressing

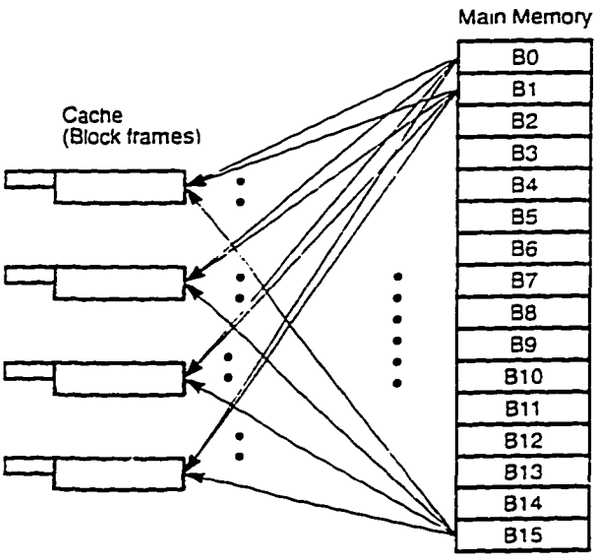


(b) Block B_j can be mapped to block frame if $i = j$ (modulo 4)

FIGURE 1.3: DIRECT MAPPED CACHE ORGANIZATION.

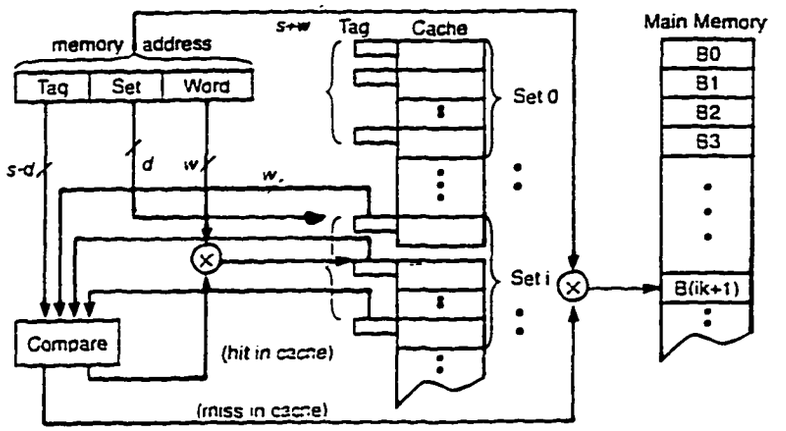


(a) Associative search with all block tags

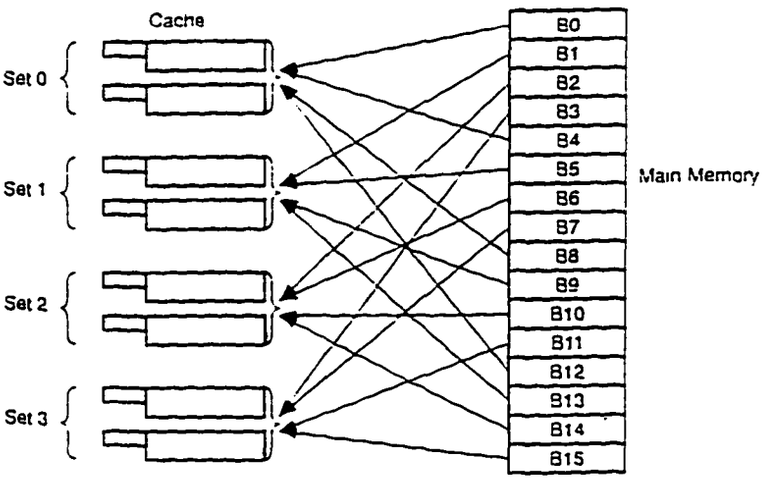


(b) Every block is mapped to any of the four block frames identified by the tag

FIGURE 1.4: FULLY ASSOCIATIVE CACHE ORGANIZATION.



(a) A k-way associative search within each set of k cache blocks



(b) Mapping cache blocks in a two-way associative cache with four sets

FIGURE 1.5: SET ASSOCIATIVE CACHE ORGANIZATION.

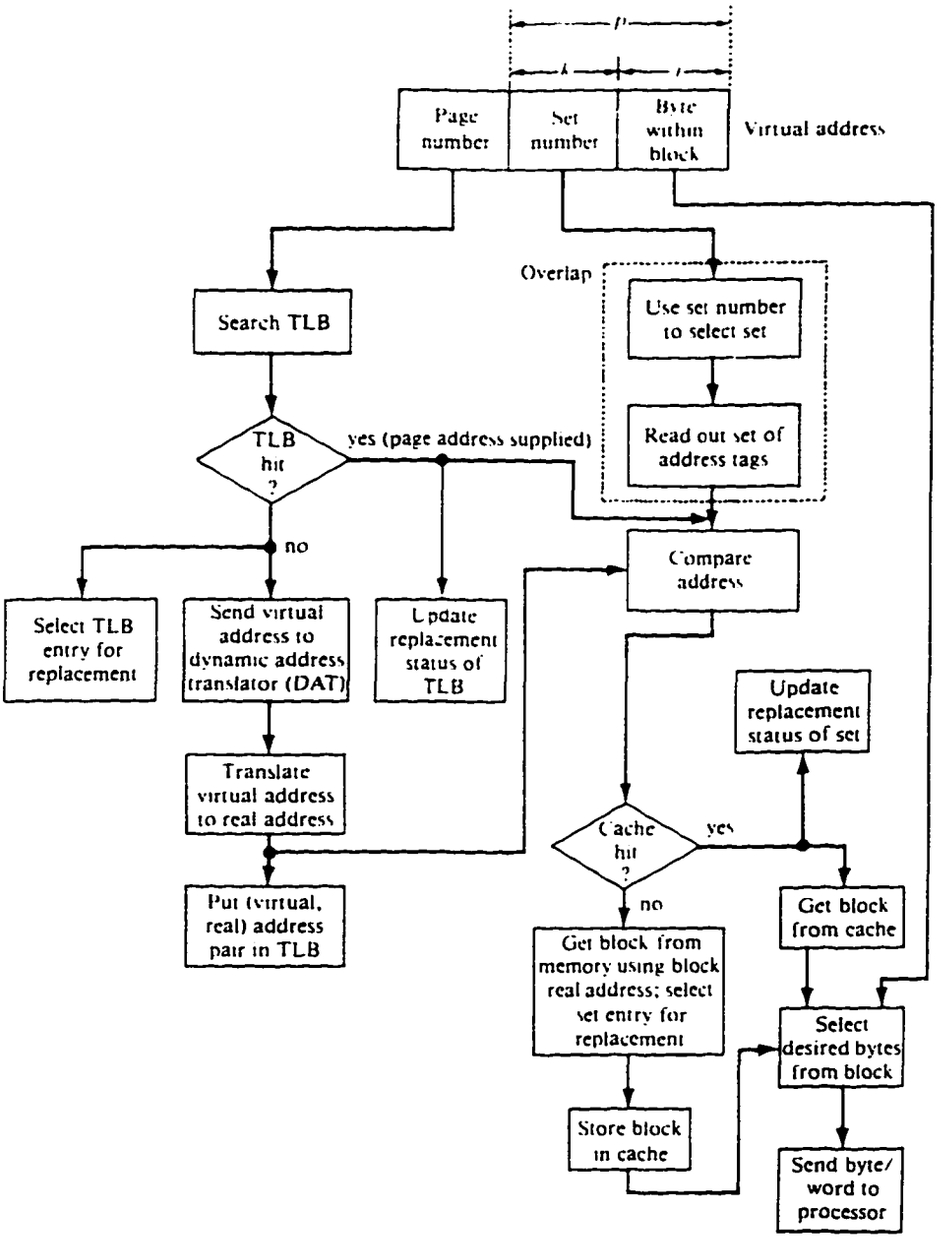


FIGURE 1.6: PARALLEL ACTIVITIES ALLOWED IN A SET ASSOCIATIVE CACHE.

CHAPTER 2

PROBLEM ANALYSIS AND REFORMULATION

2.1 Conventional Cache Replacement Algorithms

Before enunciating the popular conventional replacement algorithms, let's look at a few definitions.

Let $A(t)=a(0)\ a(1)\ a(2)\ \ldots\ldots\ldots\ldots\ a(t)$ be the given address reference string in a trace file. Let $L_t(s_i)$ be the line replaced from a resident set $S_i(t)$ on a cache miss for i=0,1,2, ..., N-1. For a N-set cache having associativity A, we define three parameters:

(a) *Forward distance* $f_t(x)$ at time t for line x is the distance of the first reference to x after time t. That is,

$$f_t(x) = \begin{cases} d, & \text{if } a(t+d) \text{ is the first occurrence of } x \text{ in } A(\infty)-A(t) \\ \infty, & \text{if } x \text{ does not appear in } A(\infty)-A(t) \end{cases}$$

(b) *Backward distance* $b_t(x)$ at time t for line x is the distance to the most recent reference of x in A(t). That is,

$$b_t(x) = \begin{cases} d, \text{ if } a(t-d) \text{ is the last occurrence of } x \text{ in } A(t) \\ \infty, \text{ if } x \text{ never appeared in } A(t) \end{cases}$$

(c) Historical distance $h_t(x)$ at time t for line x is the distance to the most recent arrival of x in set $S_i(t)$. That is,

$$h_t(x) = \begin{cases} d, \text{ if } a(t-d) \text{ is the most recent arrival of } x \text{ in } S_i(t) \\ \infty, \text{ if } x \text{ never arrived in set } S_i(t) \end{cases}$$

Using the definitions provided above, we will now state the conventional algorithms:

(1) OPTIMAL ALGORITHM (OPT): OPT replaces the line in $S_i(t)$ with largest forward distance:

$$L_t(S_i) = y \quad \text{iff } f_t(y) = \max [f_t(x)]; \quad \forall x \in S_i(t) \qquad (2.1)$$

OPT is a lookahead line replacement algorithm and is not realizable. However, it provides us with a benchmark on which we can measure the relative performance of practical algorithms.

(2) LEAST RECENTLY USED ALGORITHM (LRU): LRU replaces the line in $S_i(t)$ with the largest backward distance:

$$L_t(S_i) = y \quad \text{iff } b_t(y) = \max [b_t(x)]; \quad \forall x \in S_i(t) \qquad (2.2)$$

(3) FIRST IN FIRST OUT ALGORITHM (FIFO): FIFO replaces the line that has been in cache for the longest time:

$$L_t(S_i) = y \quad \text{iff } h_t(y) = \max [h_t(x)]; \quad \forall x \in S_i(t) \qquad (2.3)$$

Like LRU and MRU, FIFO is also a non-lookahead line replacement algorithm.

(4) MOST RECENTLY USED ALGORITHM (MRU): MRU replaces the line in $S_i(t)$ with the smallest backward distance:

$$L_t(S_i) = y \quad \text{iff } b_t(y) = \min [b_t(x)]; \quad \forall x \in S_i(t) \qquad (2.4)$$

2.2 Problem Reformulation and Foundations for Novel Algorithms

The problem of line replacement in caches has already been stated in section 1.2 of the thesis. Before this problem is analyzed and studied in an attempt to develop an understanding of the underlying phenomenon, which will eventually lead to new and improved replacement algorithms, the justification for the efforts needs to be evaluated. In order to evaluate the existence of a replacement algorithm that performs better than the popular LRU algorithm, we need to compare LRU's performance with that of the OPT algorithm (lookahead) for a given workload. A large difference in performance of the two strategies, typically 10% to 30% [21], would suggest that there exist a possibility of a practical replacement scheme which could outperform LRU. The preceding assertion is made due to the following reasons:

(1) A small difference in performance between LRU and OPT (which looks only forward) suggests that LRU (which looks only backwards) has exploited almost all the locality that was present in the reference stream.

Hence, there would be no need to develop sophisticated algorithms predicting future references.

(2) Timing constraints imposed on practical caches allow for only hardware implementation of any replacement scheme. Sophisticated hardware circuits offset cache performance by increasing the cache cycle time. This means that the performance of a cache, in terms of miss ratio, with a newly developed algorithm should be such that a slight offset in cycle time does not degrade the overall cache performance.

(3) Hardware cannot have perfect knowledge of the future, this suggests that no practical algorithm exist that could perfectly bridge the performance gap between LRU and OPT. Thus, a large gap in the performance of LRU and OPT should exist before we pursue any investigation related to the development of newer practical replacement algorithm(s).

In section 4.5 trace-driven simulation results are presented. The results indicate that in most cases OPT outperforms LRU by a considerable margin. This means that we can indeed develop algorithms that could

outperform LRU. The next question is how can we develop such algorithms ?

One of the ways to develop new algorithms that can outperform LRU is to look at the drawbacks of LRU and investigate what address reference sequences are not suitable for replacements with LRU and why ? Another approach to developing new and improved replacement algorithms is based on continuous monitoring of the two *stacks* of lines, one managed by LRU and the other by OPT, in an effort to investigate their relative behavior. From now on, a cache set will be referred to as the stack since it is basically a stack of lines ordered in accordance with the rules of a replacement policy. The experiment with stacks is expected to provide clues leading to the understanding of the phenomenon that is responsible for the success of OPT over LRU. The second approach is preferred over the former due to the following reasons:

(1) Some sequences for which LRU does not work well may not be representative of a true computer workload. This implies that the newly constructed algorithms based on the first approach may not be efficient in the real

environment.

(2) The second approach is more practical since simultaneous monitoring of OPT and LRU stacks side by side during simulations, with address traces from real computer workload, not only allow us to understand the relative behavior of the aforementioned stacks, but, it also provide hints for creating and developing new practical replacement strategies that could mimic OPT decisions.

The second approach was chosen in this work. Several unified caches (Instruction plus data cache) were simulated and studied. Associativity was varied, for the set associative caches, in the range 2 to 16. Caches with number of sets equal to 32, 256, and 2048, with an 8 byte line size was used for all simulations. Two identical groups of caches were simulated, one managed by LRU and the other by OPT line replacement algorithm. Note that the important parameter varied was the associativity of a cache; this was done because, we are interested in the behavior of a cache that is predominantly affected by the line replacement policy.

Averages were taken over all the references in the individual trace files, the sets within a cache of a given size, and three values of cache sizes.

Inputs to the cache simulator were address traces from the two trace files, TPC.DATA and BI.DATA, described in section 4.1. These trace files contain memory addresses that were collected during the execution of benchmark programs. In order to look at the relative behavior of two stacks, managed separately by LRU and OPT, we monitored two types of lines during our simulations:

(1) Lines removed from OPT stack that remained in the LRU stack.
(2) Common live lines in the OPT and LRU stacks.

The justification for doing as such is provided in the following discussion. Recall that, an OPT algorithm removes those lines from its stack which are referenced furthest into the future in order to make room for the incoming lines. This means that the displaced lines from the OPT stack should essentially be dead/inactive lines. A dead line is defined by Stone [21] as a block of information that has to exit a cache before it is

referenced again. In order to find a dead line from simulations, the simulator scans ahead on the trace file and determine if the line just referenced will be in cache on its next reference. A live line is in the cache on its next reference, whereas, a dead one is not. Figure 2.1 illustrates the activity for a typical line entering a cache. A line enters the cache at time t_{in}, gets referenced for a certain period of time $t_{last\ ref} - t_{in}$, then gets dormant (dead) for $t_{out} - t_{last\ ref}$, and it is finally discarded from the cache (t_{out}). By definition, a dead line can also exist in an OPT stack. A line deleted from the OPT stack is definitely a dead line in the LRU stack if it exists and a hit to such a line does not occur. Therefore, if it could be proved that a *miss* in OPT stack cannot result in a *hit* in LRU stack, then a line removed from OPT stack is definitely a dead line if it exist in the LRU stack. As a corollary to the preceding statement, it can be shown that OPT always provide better results than LRU.

THEOREM 2.2.1: For a given address reference, a *miss* in an OPT stack cannot have a corresponding *hit* in the LRU stack.

PROOF:

To prove the assertion, consider OPT and LRU stacks each containing *A* elements, where *A* is the associativity of a set associative cache. Without loss of generality, assume that the contents of two stacks are initially the same. A newly referenced line l_{new} will not change the contents of the two stack, if it is a hit. A hit means that the new reference l_{new} is present in both the stacks since, the two are equal. The only way the stacks will become different is when a miss on l_{new} occurs. In such a situation LRU will discard l_{LRU} and OPT will discard l_{OPT}. Where, l_{LRU} may or may not be the same as l_{OPT}.

The OPT algorithm discarded l_{OPT} because, out of *M* total references (where, *M* ---> ∞ for a general case) at least *A* distinct references have to occur (to the set/stack under consideration) before l_{OPT} is referenced again, if it is referenced any longer. Therefore, at least *A* distinct lines have to arrive in a cache set (including the presently referenced line) before l_{OPT}. At

this juncture, we would like to state that in certain instances there is a possibility for several lines in a cache to be candidates for replacement at the same time; in such cases, the replacement is essentially done randomly. The instances mentioned here refer to the following cases:

(1) The end of reference string is reached for a given trace file.
(2) Lines are not referenced in the future.
These two cases are not very important in practice because, they do not affect the miss ratios in any significant way.

Now let's return from the brief digression and note that if l_{LRU} is identical to l_{OPT} then the two stacks will remain the same. This means that, in such a case, LRU would essentially be making the same replacement decision as OPT. However, if $l_{LRU} \neq l_{OPT}$ then, the arriving A distinct lines in the future (including the line presently referenced) would definitely flush out l_{OPT} in the LRU stack irrespective of the position of l_{OPT} in the stack. For each incoming line, l_{OPT} will move down one position in the LRU stack until it reaches the end

of the stack where, it is finally removed. l_{LRU} may or may not be referenced again before it leaves the OPT stack. This means that for l_{LRU}, present in OPT stack only, two cases are possible:

(1) A hit in the OPT stack accompanied with a miss in the LRU stack.
(2) l_{LRU} is deleted after a certain period of time from the OPT stack.

Hence, this proves that for a given identical initial condition of the two stacks, a situation that results in a miss in OPT stack and a hit in LRU stack can never occur.

Q.E.D.

Thus, a line removed from the OPT stack that is present in the LRU stack is a dead line since, it would not be referenced as long as it remains in the LRU stack. As a corollary to the theorem 2.2.1, it is trivial to prove that OPT will always provide better results than LRU.

COROLLARY 2.2.2:
Consider all possible situations or states that LRU and

OPT stacks can be in, on any given reference. The four states for LRU and OPT stacks are: MISS-MISS, HIT-HIT, MISS-HIT and, HIT-MISS respectively. From theorem 2.2.1, we know that the situation that results in the case of HIT-MISS can never occur. This implies that the upper bound on the LRU's performance is dictated by the performance of OPT.

Thus far, the most important conclusion is that a line replaced from the OPT stack is a dead line in the LRU stack (if it exists in LRU stack) because, no hits can ever occur to such lines before they are discarded from the cache. The identification and replacement of such dead lines in a set associative cache is of paramount importance. These are the lines that our algorithms should be able to displace efficiently in order to achieve near-optimal performance.

Let's look at how many such dead lines exist in a typical cache ? Simulation results for the distinct dead lines (different from the ones in OPT stack) in different LRU stacks are presented in figures 2.2 and 2.3. Figure 2.2 illustrates the average number of dead lines for BI.DATA trace file. It indicates that

approximately 24.5%, 22.5%, 25%, and 24.38% of lines are dead in caches having associativities 2, 4, 8, and 16 respectively. For TPC.DATA, the approximate number of dead lines were found to be: 24%, 20%, 21.25%, and 21% respectively. TPC.DATA shows a relatively better performance than BI.DATA. The mean for all the cases comes out to be approximately 22.7%; which implies that on average nearly 1/4 of LRU stack is made up of dead lines that are absent in OPT stack !!!

During the execution of our simulation routines, we also recorded the live lines that were common in LRU and OPT managed caches. Figures 2.4 and 2.5 shows average number of common live lines in LRU and OPT stacks for different cache sizes. On average 60.85% of live lines are identical in the two stacks. This means that LRU and OPT stacks are not very much different hence, LRU may be used as a basis for constructing new and improved algorithms.

2.3 Conclusions

From the discussion provided in this chapter, we can draw three major conclusions:

(1) Near-optimal cache replacement algorithms are feasible.

(2) Line replacement problem in caches can be reformulated as the one related to the recognition of live and dead lines. This is a suitable problem for neural networks which are known to have produced excellent results for similar dynamic data sets, and similar optimization and prediction problems [194]-[201].

(3) Since 75% of the lines in LRU and OPT stacks are identical, an attempt can be made to modify LRU algorithm to develop new near-optimal algorithm(s) that would efficiently delete dead lines from the caches.

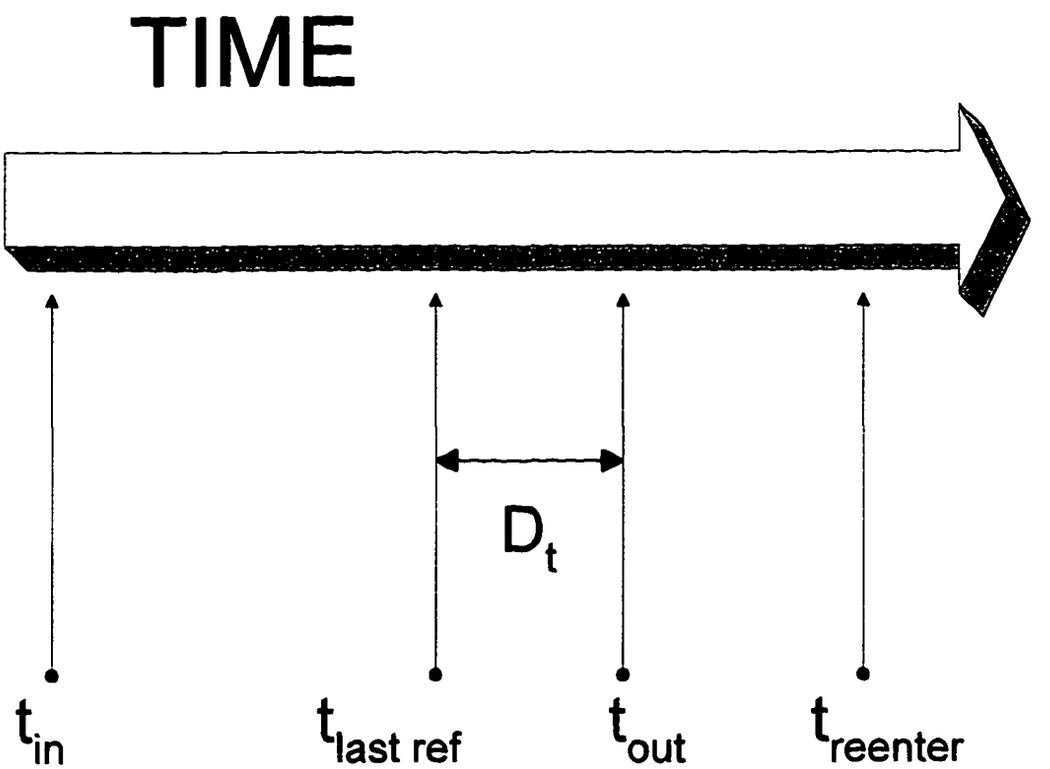


FIGURE 2.1: ACTIVITY FOR A TYPICAL CACHE LINE

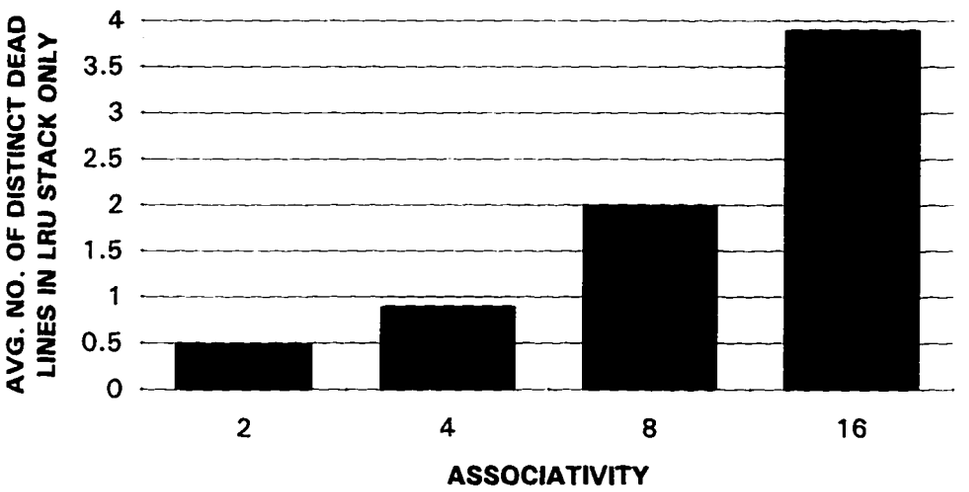


FIGURE 2.2: AVERAGE NUMBER OF DISTINCT DEAD LINES PRESENT IN LRU STACK ONLY vs. THE ASSOCIATIVITY OF A CACHE FOR BI.DATA TRACE FILE (NO. OF SETS = 32, 256, 2048 AND LINE SIZE = 8 BYTES)

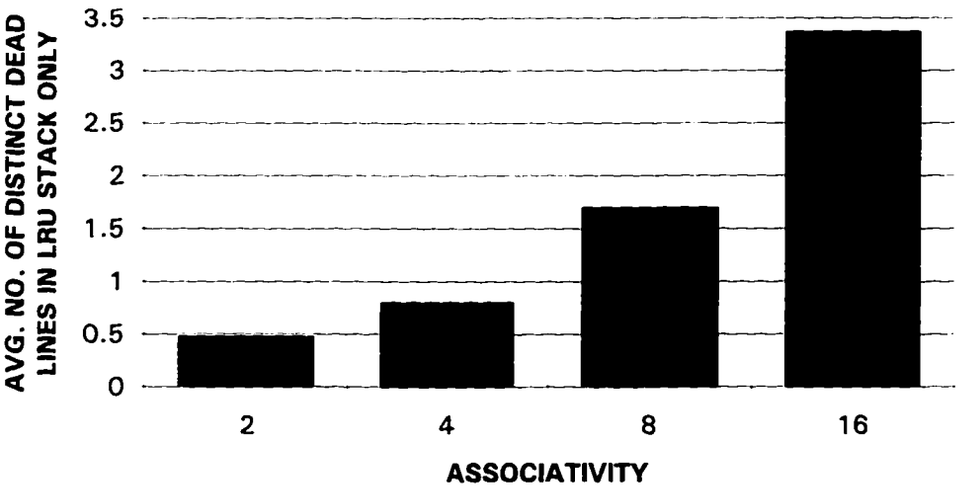


FIGURE 2.3: AVERAGE NUMBER OF DISTINCT DEAD LINES PRESENT IN LRU STACK ONLY vs. THE ASSOCIATIVITY OF A CACHE FOR TPC.DATA TRACE FILE (NO. OF SETS = 32, 256, 2048 AND LINE SIZE = 8 BYTES)

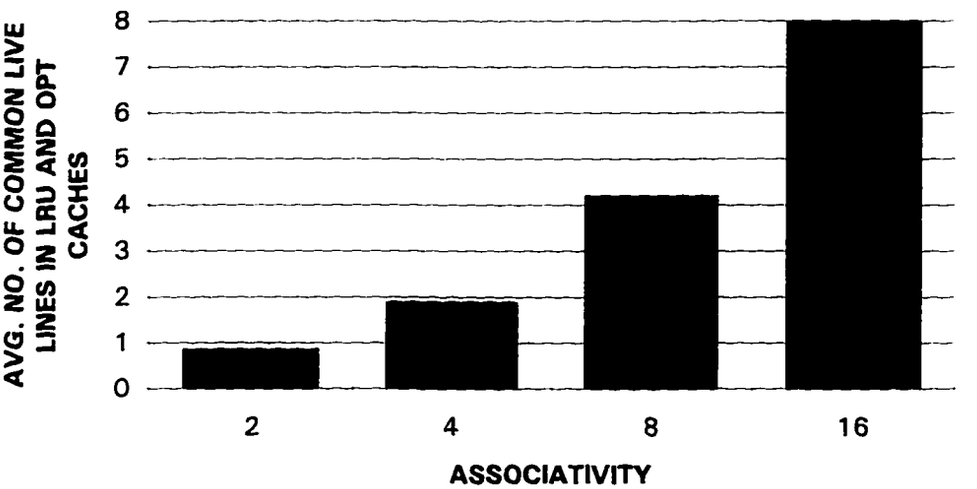


FIGURE 2.4: AVERAGE NUMBER OF COMMON LIVE LINES IN LRU AND OPT STACKS vs. ASSOCIATIVITY FOR BI.DATA TRACE FILE (NO. OF SETS = 32, 256 ,2048 AND LINE SIZE = 8 BYTES)

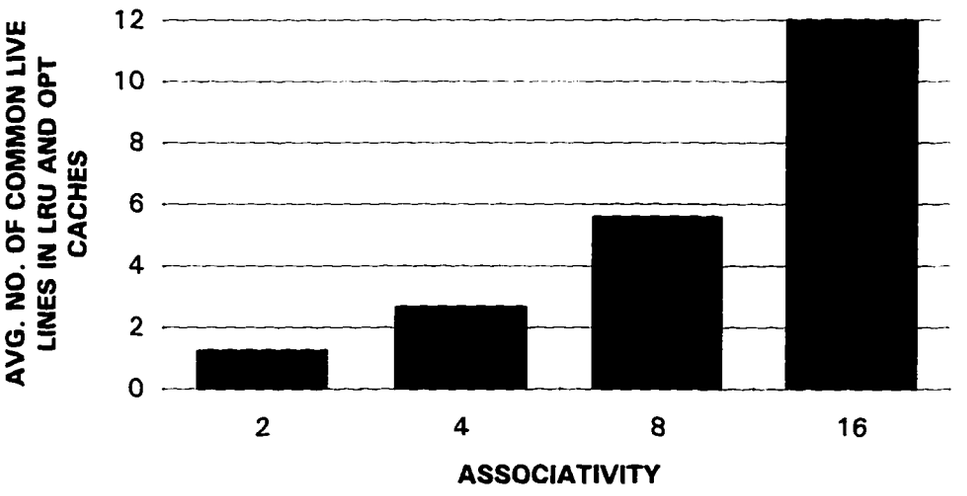


FIGURE 2.5: AVERAGE NUMBER OF COMMON LIVE LINES IN LRU AND OPT STACKS vs. ASSOCIATIVITY FOR TPC.DATA TRACE FILE (NO. OF SETS = 32, 256, 2048 AND LINE SIZE = 8 BYTES)

CHAPTER 3

NEA AND EA CACHE REPLACEMENT ALGORITHMS: THEORY, ALGORITHMS, AND IMPLEMENTATION

3.1 Introduction to Neural Networks

The study of neural networks (NNs) is an attempt to simulate and understand the nervous system. In particular it is of interest to define an alternative computational form that attempts to mimic the brain's operation in several ways. Interest in neural networks, in the last few years has increased due partly to a number of significant break-through in research on network types and operational characteristics. Advances in the computer hardware technology that made neural network implementation relatively faster and more efficient has also provided boost to the progress in neural networks. Much of the drive has arisen because of numerous successes achieved in demonstrating the ability of neural networks to deliver simple and powerful solutions, particularly in the fields of learning and pattern recognition, both of which have proved to be difficult areas for conventional computing.

Digital computers provide a media for well-defined numerical algorithms processing in a high performance environment. This is in direct contrast to many of the properties shown by biological neural systems, such as: creativity, generalization, and understanding. However, computer-based neural networks at present provide a move forward from digital computing in the direction of biological systems. Because of its inter-disciplinary nature, encompassing computing, biology, and neuropsychology etc., the field of neural networks attracts a variety of interested researchers from a broad range of backgrounds. The area has become very exciting and interesting due to massive new ideas still to be investigated, tested, and put to appropriate applications. Today, there are many different types/models/paradigms of neural networks, reflecting the emphasis and goals of different research groups. Each network type, generally, has several variations which address some weakness or enhance some feature of the basic network.

The present field of neural networks links a number of closely related areas, such as: parallel and distributed computing, connectionism, and neural computing. These

areas are brought together with the common theme of attempting to exhibit the method of computing which is witnessed in the study of biological neural system. This has lead to an interesting definition of neural computing [202]; Neural Computing is a study of networks of adaptable nodes which, through a process of learning from task examples, store experiential knowledge and make it available for use. Although, the definition represents a significant leap from standard digital computing, it does indicate a rather limited practically oriented state at the present time, leaving behind many areas to be covered towards the artificial reconstruction of complex biological systems.

A fundamental aspect of artificial neural networks is the use of simple processing elements which are essentially models of neurons in the brain. The brain is composed of approximately 10^{11} neurons of many different types. A neuron is the fundamental cellular unit of the nervous system and, in particular, the brain. Each neuron can be regarded as a simple processing element which receives and combines signals from many other neurons through input structures called Dendrites. For a combined input signal having a values greater than a

certain threshold, the firing of neuron takes place resulting in an output signal that is transmitted along a cell component called Axon. The Axon of a neuron splits up and connects to Dendrites of other neurons through a junction referred to as a Synapse. The strength of signal transmitted across a Synapse is called Synaptic Efficiency, which is modified as the brain learns.

In artificial neural networks, the element that corresponds to a biological neuron is called a processing element (PE). A simple PE combines its input paths by adding up the weighted sum of all its inputs. The output of a PE is the signal that is generated by applying the combined inputs to an appropriate transfer function. Learning takes place in the form of adjustment of weights connecting the inputs to the PE. There are several ways in which the PEs are interconnected in neural networks as groups called layer, and also, there exist a variety of learning rules that determine how, when, and by what magnitude are the weights updated. The direction of signal flow, mode of learning, type of transfer function, values for learning parameters, and the number of PEs in each layer etc., are few of the

topics which are actively researched and experimented with, at the present time, in order to develop neural networks suitable for applications in diverse areas such as: financial prediction, signal analysis and processing, process control, robotics, classification, pattern recognition, filtering, optimization, prediction, speech synthesis and recognition, and medical diagnosis.

Active research in neural networks by a group of scientists, engineers, and scholars has resulted in a large array of network types. There are several different ways in which these networks can be characterized/categorized/divided. In this thesis NNs have been divided into two categories, namely: estimating type and the non-estimating type. The NNs that use Parzen estimators for the estimation of probability density function (pdf) for a given data set are labeled as the estimating type of NNs. All the other neural networks that fall into the other category are, accordingly, called non-estimating type of NNs.

In the next two sections of this chapter, adaptive algorithms based on different categories of neural

networks are developed to perform predictions in order to identify cache lines to be replaced (dead cache lines as opposed to live lines) for memory references that result in cache misses. The NNs used by the algorithms remain in the learning mode for our real-time application. For further study on the theory and applications of NNs, several good references are available [203]-[252].

3.2 Non-Estimating Neural Networks Based Algorithm for Adaptive Cache Replacement [NEA]

In order to eliminate dead lines, the main task is to design a NN-based predictor. The job of our NNs is to learn a function that may approximate the unknown relationship between the past and future values of addresses.

f : past → future ; ∀ (past, future) ∈ Domain of addresses

The function f can be learned using the tag field of the addresses. Recall, that a set-associative cache partitions the addresses into tag, set, and the word fields. Where, the word field is used to select a word

within a cache block frame/line and the set field is used for indexing. The tag field preserves the mapping between the cache and the main memory. This means that the development of f with the tag field cannot be done without using the set field for indexing a group of neurons in NNs.

Here, the function f actually represents a *predictor*. The parameters of such a predictor can be developed through the use of statistical properties of data such as: mean, variance, probability density functions (pdf), cumulative distribution functions (cdf), and power spectral density etc. This means that our NNs would behave as specialized predictors if they are provided with some knowledge about the statistical properties of the tags [198]. The problem with this approach is that tags, like other similar data sets, do not have well-defined statistical properties. To accomplish the task, NNs could be trained to learn the *histogram of references* (an estimated pdf is called a histogram). Initially, the performance of the predictor is expected to be low, however, the performance improves as the histogram develops. A well-developed histogram may contain sufficient locality information about the

address references and it can be used for guiding the replacement decisions. In order to accomplish this with NNs, we need at least two tags at a time, T(i) and T(i+1), for the training/learning. This implies that the network learning should be delayed until a new tag arrives. A tag T(i) is a vector that is made up of 0's and 1's. The size of each vector T(i) depends on the computer architecture. For example, for a Q-bit address machine that incorporates a cache with N sets and L words per line, the size of the vector T(i) would be equal to $Y=(Q-(\log_2 N + \log_2 L))$ bits. Thus, a Y-dimensional vector T(i) should be applied to the input layer of NNs (predictor) and the output $\hat{T}(i+1)$ (estimate of T(i+1)) is compared with T(i+1) in order to get error values for feedback purposes. The repetition of the procedure will eventually lead to the development of predictor parameters containing histogram information of the addresses.

The above discussion gives some insight into the algorithm that would make NNs behave as predictors of address references. The important question that still remains is that how can the predictor information be used for line replacements or dead line identification

in a cache in real time ? This question is answered as follows: What we want to do is to make NNs learn the histogram of addresses. At the same time, we would also like to look at the correlation of tags with the pdf of references stored in the neurons (or PEs). Such a correlation will provide information regarding the similarity of the incoming addresses with the previous addresses (stored as the histogram of references) associated with the A lines belonging to a particular cache set. A low value for such a correlation with any of the cache lines within a set i means that the referenced address hasn't been seen, relatively recently, at that particular line position/slot for the set i. Thus, for a low correlation value, the referenced address can be recognized as the one belonging to a transient type of cache line with respect to the lines that have previously arrived at the particular slot in set i.

Since such an incoming transient line is significantly apart from the lines that have arrived earlier at the slot (as indicated by the low correlation value) therefore, according to the Temporal and Spatial localities of references (defined in section 1.4), the

line that is currently occupying the slot in set i (if any) can be regarded as a dead line. Hence, the important parameter that is used for guiding the replacement decision is not $\hat{T}(i+1)$, rather, it is the correlational response of a *particular layer of neurons containing pdf of referenced addresses*. The number of such neurons in the layers, should obviously be equal to the associativity of the cache in order to be able to identify the cache line to be replaced within the set. The task, therefore, is to select a neural network architecture and train it to learn the histogram in such a way so that at least a layer of neurons in NNs could provide the desired information. The responses of these neurons to an incoming tag T(i) will, therefore, determine the extent to which a tag vector T(i) is seen in the past. This can be accomplished by correlating the pdf of references, corresponding to each cache line in a set, with the present tag vector T(i).

Consider the finite correlation of two sequences x_n and y_n,

$$r_n=\sum_{k=0}^{m} x_k y_{k-n}=x_0 y_{-n}+x_1 y_{1-n}+\ldots+x_m y_{m-n} \qquad (3.1)$$

The NNs can compute the sum as follows for an input

vector X and the weight vector W :

$$X^T W = x_0 w_0 + x_1 w_1 + \ldots + x_m w_m \qquad (3.2)$$

Equating the right hand side of equations (3.1) and (3.2), we get the following values:

$$w_o = y_{-n},\ w_1 = y_{1-n},\ \ldots\ldots,\ w_m = y_{m-n}$$

This means that for certain weight values, the inner product computed by neurons, can in fact provide the correlation information. With this information, we can determine which line is the most suitable candidate for replacement in a cache set.

For an associativity A, we are looking towards an A-class problem for which we require

$$\max_{i=1,\,2,\,\ldots,A} \| X - W_i \|$$

Expanding $\| X - W_i \|$ we get,

$$\| X - W_i \| = [(X-W_i)^T (X-W_i)]^{1/2} = (X^T X - 2X^T W_i + W_i^T W_i)^{1/2} \qquad (3.3)$$

From (3.3) it is obvious that for max $\| X - W_i \|$, i=1,2, ..., A, we require min $(X^T W_i)$. Therefore, based on the guidelines provided by the locality of reference property for approximating dead lines [21], the most suitable line to be replaced is the one that corresponds to the neuron (in a layer) with the smallest output response. Thus, the identification of such a layer

becomes imperative for our job. The task is a little complicated in the case of the non-estimating type of NNs (BPNN, MNN, LVQ, RBFN). This is because, we need to train such NNs to develop the histogram by externally providing T(i+1). In this research, a heuristic solution is suggested for the problem of identifying a layer of neurons that could provide the replacement information. Trace-driven simulation experiments indicate that by inserting a low-dimensional layer of neurons between two identical NNs (Interface layer), the problem of identifying the appropriate cache line to be replaced could be resolved. The above discussion leads to the line replacement algorithm described by the following steps (containing appropriate pseudo code):

ALGORITHM-I

This algorithm is developed for replacements with non-estimating type of neural networks like: Backpropagation Neural Networks [BPNNs], Modular Neural Networks [MNNs], Radial Basis Functional Networks [RBFNs], and Learning Vector Quantization Networks [LVQs].

Step1: (Partitioning Of NNs)

Partition the NNs into groups. Each group/neural network

module deals with a specific set. This means that for N sets of a cache the algorithmic space/time complexity would be O(αN). Where, α is a factor that relates to the complexity of NNs within a module.

Pseudo code:

```
Start()
{ for i ← 0 to N-1
Module [i] ← Neural_Network (desired neural network
                                specifications)
comment1: N=total number of sets in a set associative
          cache.
comment2: desired neural network spec.= {total number of
neurons, learning rules, learning rates, momentum and,
transfer functions}
}
```

Step2: (Partitioning of Modules)

Each module of neural network is partitioned into two sub-modules. The first sub-module should have an input layer and one or more hidden layer(s). The second sub-module should have one or more similar hidden layer(s) and an output layer. The two sub-modules are to be connected by a lower dimensional layer of NNs, the

interface layer, containing A neurons (A=associativity of a cache) that can provide the desired response for replacements.

Pseudo code:

```
Module_Partitioning ()
{ input_layer_size ← tag_size
   hidden_layer1.1_size ← variable 1
   hidden_layer1.2_size ← variable 2
   hidden_layer2.1_size ← variable 3
   hidden_layer2.2_size ← variable 4
   output_layer_size ← tag_size
comment: At most two hidden layers are used in the
experimentation. The two hidden layers are represented
by X.1 and X.2 in the pseudo code.
   interface_layer_size ← Associativity
for j ← 0 to N-1
 { network1 ← Sub_Module1 (Module[j], input_layer_size,
            hidden_layer1.1_size, hidden_layer1.2_size)
  network2 ← Sub_Module2 (Module[j], output_layer_size,
            hidden_layer2.1_size, hidden_layer2.2_size)
  network3 ← Interface_Layer (Module [j], Associativity)
```

```
  neural_network [j] ← network1 [LINK] network3 [LINK]
                       network2
  }
}
```

Step3:(Initialization)

Initialize all the weights in NNs to small random values very close to zero.

Pseudo code:

```
Random_Weights ()
{ for sets ← 0 to N-1
neural_network[sets].weights ← Small_Random_Numbers
                               (sets)
}
```

Step4:(Module Selection)

Partition each incoming address into tag, word, and set fields. The set field is to be used for selecting one of the NN modules.

Pseudo code:

```
reference ← Get_Reference (Trace File)
```

```
Extract_Fields()
{ line_bits ← Convert_Into_Bits (line_size)
  set_bits ← Convert_Into_Bits (N)
  tag_bits ← (reference_bits) - (line_bits + set_bits)
  word_number ← Get_Field (reference, line_bits)
  set_number ← Get_Field (reference, set_bits)
  tag_field ← Get_Field (reference, tag_bits)
}
```

Step5:(Replacement Line Identification)

Apply the input Y-dimensional tag vector T(i) to the selected module *n* and look at the response from the interface layer. The lowest response from neuron i, i=1, 2, ..., A, indicates that the i^{th} cache line within the set *n* should be replaced on a miss. On a hit situation or when a cache line is empty (illegal tag), the response from the NNs should be disregarded.

Pseudo code:

```
if ( Miss() == True  &&  all_cache_lines_full
                                    [set_number] == True )
then
```

```
{ neural_network [set_number].input_layer ← tag_field
  neuron_number [set_number] ← Lowest_Response (
            neural_network[set_number].interface_layer)
for i ← 0 to (interface_layer_size - 1)
{ if (i == neuron_number [set_number])

then  control_unit_field [set_number][i] ← +1.0; replace
                                                  line i
else   control_unit_field [set_number][i] ←  - 1.0;
                                    do not replace line i
 }
}
else  control unit  ← disregard neural network response
                      from the interface layer
```

Step6: (Learning)

The output response from the second sub-module is compared with the next incoming tag T(i+1) and the resulting error $\{T(i+1) - \hat{T}(i+1)\}$ is used for the weight adjustment.

Pseudo code:

```
Learning ()
```

```
{ if (Response from the output layer of a neural network
    and the next tag field is available for a given set)
   then {
error_vector [set_number] ← ( Response (neural_network
           [set_number].output_layer) - next_tag_field)
Update_Weights [set_number] ← ( neural_network
              [set_number], error_vector [set_number] )
          }
  else   curtail learning
}
```

3.3 Estimating Neural Networks Based Algorithm for Adaptive Cache Replacement [EA]

The algorithm for the replacement of cache lines using estimating type of NNs is developed by proceeding along similar path as for the non-estimating category of the algorithm. Again, the dual job of the required algorithm is: (1) To train the NNs so that they learn the histogram of the incoming data, and (2) To get the information from NNs regarding the dead line/block frame to be replaced. For the training of the estimating type of NNs, like PNNs and GRNNs, with an input Y-dimensional

tag vector T(i), our algorithm needs to partition the entire neural network into modules. This is necessary due to the following factors:

1. Non-partitioned NNs will have to use the set and the tag fields of the memory address as an input to the neural networks. This increases the dimensionality of the input vector, resulting in a stretched cycle time due to slower response of large neural networks.

2. The increment in the size of the input vector not only translates to a corresponding increase in the size of the input layer neurons, but, it also enhances the probability of error in replacing a dead line from the cache. This is because, the error is no longer constrained to within a single set for the type of algorithm that we are looking at.

3. The higher probability of error in replacing an inactive line (dead line) from the cache may undermine the usefulness of the neural network based replacement algorithms.

The learning of statistical/estimating type of NNs

with tag T(i) is done by estimating the categorical (related to a category/class) pdf, internally, using Parzen windows. By categorical pdf, we mean the histogram/pdf that is developed for each cache line or slot associated with any set i. Such pdfs are developed from the address reference stream using a non-parametric pdf estimation technique called Parzen estimation. A Parzen estimator is of the form specified by equation (4.12). The functions developed by Parzen estimator is used to estimate the likelihood of an incoming line being within the given categories/slots for any set i. Optionally, this can be combined with the a priori probability of each slot within a set to determine the most likely category for a given block frame/cache line. For the application under study, the relative frequency (a priori probability) is unknown for the given address traces, therefore, all categories are assumed to be equally likely and the determination of a category is solely based on the closeness of the tag to the distribution function of a category or a cache slot.

In order to develop correct pdfs, we need to guide the learning process of the neural networks by correctly identifying the slot within a set where the newly

arrived line is stored in the cache. This can be done if our algorithm is such that it monitors the output response from NNs as well as the response from the control unit, and feeds back the correct slot information for the cache line that needs to be placed. The preceding activity is needed to inform NNs regarding the association of tags with correct line position/slot within a set. It is important to note that the information from the control unit specifies the hit position of a line within a set/stack in the case of a cache hit. Also, it provides the EA algorithm with the location of an empty slot within a cache set. Thus, the control unit information is critical for our replacement policy since it is required for proper learning of NNs in the cases where we have cache hit or vacant line situation. The stated procedure will ensure that the history, that is stored in the neural network weights, is accurately and dynamically developed. With this strategy, we don't need T(i) and T(i+1) for the determination of feedback error. This means that the learning process is not delayed until a new memory reference arrives as opposed to the non-estimating type of NNs. The discussion leads to the following algorithm for PNNs/GRNNs based cache replacements.

ALGORITHM-II

This algorithm is developed for replacements with estimating type of neural networks like: Probabilistic Neural Networks [PNNs] and General Regression Neural Networks [GRNNs].

Step1: Same as Step1 of algorithm 1.

Step2: Same as step3 of algorithm 1.

Step3: Same as step4 of algorithm 1.

Step4:(Replacement Line Identification)

Apply the input Y-dimensional tag vector T(i) to the selected module *n* (the module is selected by the set field of the incoming address) and send the output response of the PNN/GRNN to a *transformation module*. This module transforms values as follows:

g : Lowest value → Highest possible output value for neurons

All other values → Lowest possible output value for neurons

Index corresponding to the highest output value of the transformation module represents the cache line to be replaced from the set *n*.

```
Pseudo code:
if ( Miss() == True  &&   all_cache_lines_full
                          [set_number] == True)
then {
neural_network [set_number].input_layer ← tag_field
for i ← 0 to (output_layer_size - 1)
transformation_module_input [set_number][i] ←
             Neural_Network_Response ( neural_network
             [set_number].output_layer[i])
comment: output_layer_size = Associativity
for j ← 0 to (output_layer_size - 1)
{ if (transformation_module_input [set_number][j] ==
                                       smallest_value)
  then transformation_module_output [set_number][j] ←
                                                    +1.0
  comment: Cache line j should be replaced
  else transformation_module_output [set_number][j] ←
                                                    -1.0
 }
}
```

Step5:(Learning)

The output values from the transformation unit is

compared with its input values for the error. This means that

ERROR = { g(input) - input}

However, for the hit situation or for the case when a cache line is empty, the error is evaluated by comparing the input values of the transformation module with a vector supplied by the control unit (CU) of the controller. This vector should have a value of +1.0 (highest possible unscaled output value for a neuron) at the coordinates corresponding to the hit-position (cache hit case) or tag-insertion position (empty line case). All other values should be -1.0 (lowest possible unscaled output value for a neuron).

Pseudo code:

```
if ( Miss() == True  &&  all_cache_lines_full
                          [set_number] == True)
then { error_vector [set_number] ←
          ( transformation_module_input [set_number] -
           transformation_module_output [set_number] )

Update_Weights [set_number] ( neural_network[set_number]
                          ,error_vector[set_number] )
       }
```

```
else { if ( Hit() == True)
          then { for i ← 0 to (Associativity -1)
                    { if ( i == hit_position [set_number])
        then error_vector [set_number][i] ← (+1.0 -
                              transformation_module_input)

     else error_vector [set_number][i] ← (-1.0 -
                              transformation_module_input)
                    }
Update_Weights [set_number] ( neural_network[set_number]
                              ,error_vector[set_number] )
                    }
           else { comment: empty line case begins
                    for j ← 0 to (Associativity -1)
             { if (j == empty_line_position [set_number])
        then error_vector [set_number][j] ← (+1.0 -
                              transformation_module_input)
        else error_vector [set_number][j] ← (-1.0 -
                              transformation_module_input)
                    }

Update_Weights [set_number] ( neural_network[set_number]
                              , error_vector[set_number] )
```

}

3.4 Proposed Implementation for the NEA and EA Replacement Algorithms

Based on the theory and the algorithms for different categories of NNs, two different architectures are proposed that are presented in figures 3.1 and 3.2 for the NEA and EA schemes respectively. Looking at figure 3.1 we observe that the set field of an incoming address is extracted by the set selector to identify a neural network module that contains the information related to a particular cache set. The input and the output lines for only one neural network module is shown for brevity. All the other neural network modules are identical copies of the one shown. The tag field of an address is fed in the NN module as an input through a register. The sub-modules are represented by rectangular boxes. These boxes contain input, output, and the hidden layer(s) for non-estimating type of NNs such as: BPNN, MNN, RBFN, and LVQ. The number of neurons in the input and output layers are identical and equal to the dimensionality Y of the tag vector T(i). The sub-modules are connected by

a neural network layer called the interface layer. Each neuron in the interface layer corresponds to a cache line within a set. The error values are generated by comparing the estimated tag $\hat{T}(i+1)$ with a new incoming tag T(i+1). Therefore, the error calculation for a set S_i is delayed by approximately the amount of time it takes for a new address to arrive (the new address should map to set S_i).

The error is computed through an adder/subtractor accompanied with a buffer as shown by a circle in figure 3.1. This error is fed back to the neural networks, for the set under consideration, in order to develop the pdf of addresses. The minimum selector (MIN SEL) is for identifying a neuron, in the interface layer, that provides the lowest response value and accordingly sends the line replacement information to the control unit (CU). The initial performance of this architecture would obviously be low because of the initial random weight selection. However, performance improves as NNs learn the pdf.

The hardware implementation for the PNNs/GRNNs based replacement algorithm is shown in figure 3.2. Here, only

the current value of tag T(i) is presented directly to the NN module as an input. The output layer for PNNs/GRNNs is shown external to the rectangular box in figure 3.2. The transformation module, in figure 3.2, transforms the output values from PNNs/GRNNs for two purposes:

(1) To identify the corresponding cache line to be replaced.

(2) To determine the error.

Again, the module transforms values as follows:

g : Lowest value → Highest possible output value

for a neuron

All other values → Lowest possible output value

for neurons

For example, a 4-dimensional vector $[0.1\ 0.9\ 0.8\ 0.3]^T$ is transformed as given below.

$[0.1\ 0.9\ 0.8\ 0.3]^T \xrightarrow{g} [1.0\ -1.0\ -1.0\ -1.0]^T$

A multiplexer (MUX) is used to select the vector from either the control unit (CU) or the output of the

transformation unit. The information as to which of the inputs should be selected by the MUX comes from the control unit. The lower input lines going into the MUX are selected by the CU on a hit situation or when a cache line in empty. However, on a cache miss, the upper lines (representing the desired output vector) are selected for the purpose of error computation.

3.5 Conclusions

This chapter began by presenting an overview of neural networks. Next, adaptive cache replacement algorithms were developed. In particular, we looked at the theory behind each algorithmic step. Two algorithms based on neural networks were developed (NEA and EA algorithms). One of the algorithm was designed with non-estimating category of NNs in mind. Recall, that the non-estimating type of NNs were defined as the ones that do not estimate probability density function (pdf) automatically from the data stream. The other algorithm was structured for the estimating (statistical) category of NNs. The chapter concluded with the addressing of the implementation issue.

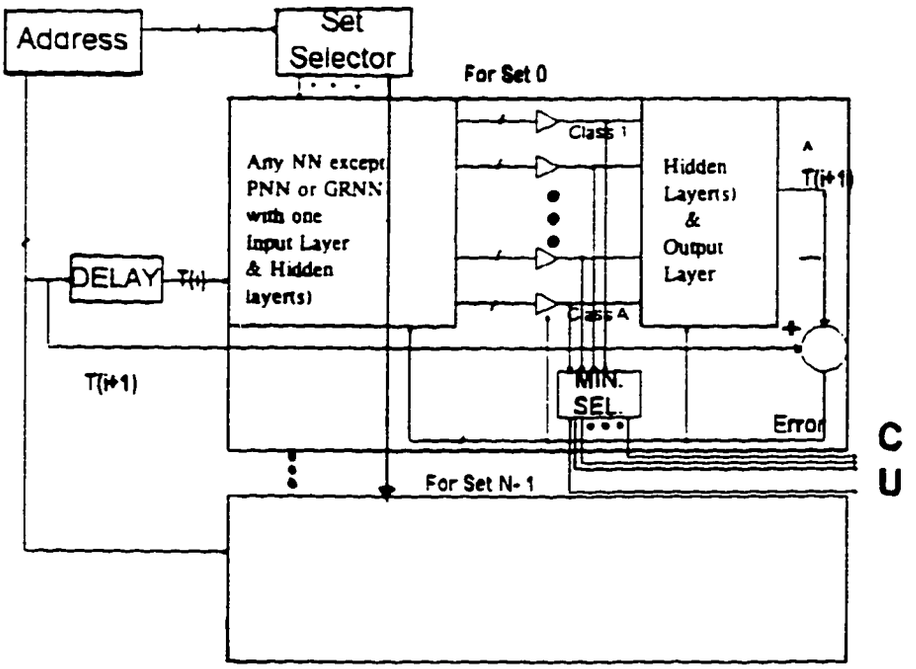


FIGURE 3.1: BLOCK DIAGRAM OF THE CACHE CONTROLLER WITH ANY NEURAL NETWORK PARADIGM EXCEPT PNN/GRNN.

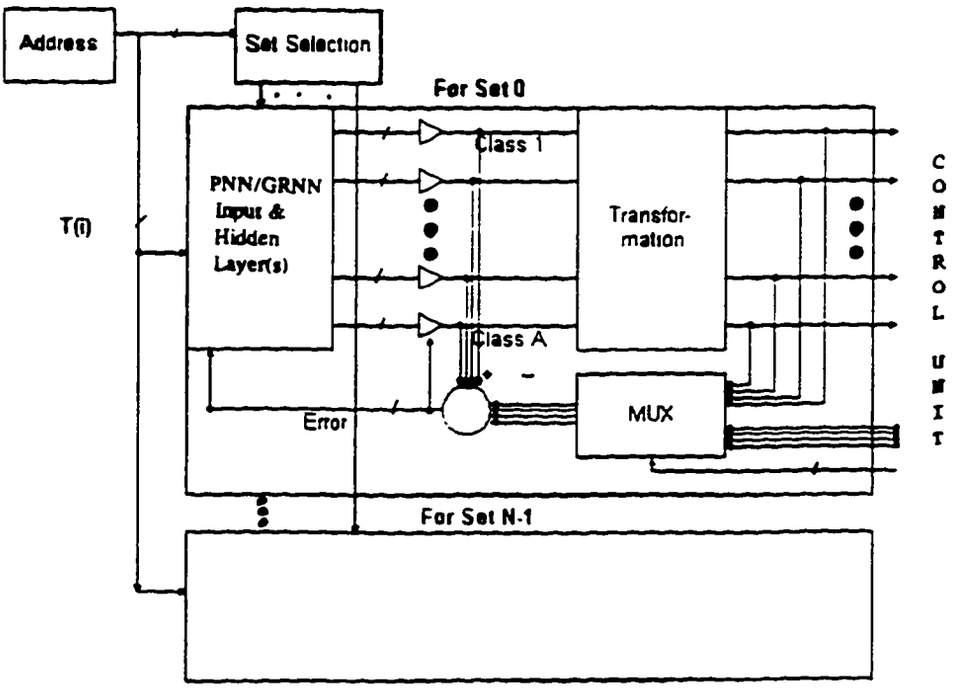


FIGURE 3.2: BLOCK DIAGRAM OF THE CACHE CONTROLLER WITH PNN/GRNN PARADIGMS.

CHAPTER 4

PERFORMANCE EVALUATION OF THE NEA AND EA REPLACEMENT ALGORITHMS FOR DIFFERENT VARIANTS OF NEURAL NETWORK PARADIGMS

4.1 Cache Simulation

In our research, we have used trace-driven simulations to accurately obtain information on the relative performance of various neural network based algorithms and the conventional algorithms. The two trace files used were named TPC.DATA and BI.DATA. TPC.DATA is a trace file that was generated on an IBM PC running Borland's Turbo Pascal Compiler. The other trace file, BI.DATA, was generated using Microsoft's Basica Interpreter. Each of these trace files contains approximately 1/2 million memory addresses. The memory references were recorded as a three byte word in little endian format and stored without any delimiters [21].

The number of sets for the caches simulated by our simulator varied from N=32 to N=2048. The line size was fixed at 8 bytes per block for the entire simulation.

Associativity A was varied and three different cases were studied (that is A=2, A=4, and A=8). Therefore, the figures illustrating simulation results are partitioned into three parts namely (a), (b), and (c), corresponding to the three values of associativity under consideration.

4.2 Performance of NEA Algorithm Based on Several Variants of Backpropagation Neural Networks

A typical backpropagation neural network (BPNN) has an input layer, an output layer, and at least one hidden layer (see figure 4.1). There is no theoretical limit on the number of hidden layers. However, in this work, we have tried to limit the number of cases under study by considering at most two hidden layers. Some work has been done which indicates that a maximum of 3 hidden layers are required to solve an arbitrarily complex pattern classification problem [198]. Neurons/processing elements (PE) are represented by a triangle in figure 4.1. A simple neuron computes the weighted sum of its inputs and accordingly generates an output. The weights on the inputs are modified in the learning process

(supervised learning) to adapt to the required mapping. Thus, a BPNN element transfers its inputs as follows:

$$O_k^{[s]} = f\left[\sum_j w_{kj}^{[s]}\, y_j^{[s-1]}\right] = f(net_k^{[s]}) \qquad (4.1)$$

Where,

$y_j^{[s-1]}$ = current output state of j^{th} neuron in layer s-1.

$w_{kj}^{[s]}$ = weight on connection joining the j^{th} neuron in layer(s-1) to k^{th} neuron in layer s.

f = activation/transfer function.

A suitable activation function can be chosen for each layer by trial-and-error method from among several commonly used functions such as TanH, sigmoid, and linear. In equation (4.1) we have used the traditional sum to obtain $net_k^{[s]}$. However, several other variants of $net_k^{[s]}$ may be used and their relative effect on the BPNN's performance can be studied. A few of the general purpose summation functions are:

$$\text{Maximum} : net_j = MAX_i\ (w_{ji}\, y_i) \qquad (4.2)$$

$$\text{Minimum} : net_j = MIN_i\ (w_{ji}\, y_i) \qquad (4.3)$$

$$\text{Majority} : net_j = \sum_i sgn\ (w_{ji}\, y_i) \qquad (4.4)$$

$$\text{Product} : net_j = \prod_i w_{ji}\, y_i \qquad (4.5)$$

$$\text{City-Block} : net_j = \sum_i \left| y_i - w_{ji} \right| \qquad (4.6)$$

The information propagates through BPNN in response to the input patterns. Differential error at each hidden layer is computed as follows:

$$\delta_j^{[s-1]} = \frac{-\partial E}{\partial net_j^{[s-1]}} = f'(net_j^{[s-1]}) \cdot \sum_k (\delta_k^{[s]}\, w_{kj}^{[s]}) \qquad (4.7)$$

and the corresponding delta weights are added to all the weights in the system.

$$\Delta w_{ji}^{[s-1]} = \text{lcoef} \cdot \delta_j^{[s-1]} \cdot z_i^{[s-2]} + \text{momentum factor} \qquad (4.8)$$

Where, lcoef= learning coefficient

This is done for each (input, desired output) pair when delta-learning rule is in effect. However, the convergence speed for BPNN can be improved when other rules such as normalized-cumulative-delta (Norm-Cum), Delta-Bar-Delta (DBD), and Max-Prop. rules are used along with the momentum factor. These learning rules are used in our experimentation.

Figures 4.2 and 4.3 provide the miss ratio information for different variants of BPNN paradigm with BI.DATA and TPC.DATA as the trace files, respectively. T's assigned to the legends denote the type of BPNN. Complete description of T's is given in Appendix B. Observe from the figures 4.2 and 4.3 that BPNN of type T6 has a relatively better performance than the other types. This means that the Norm-Cum learning rule along with the TanH transfer function is suitable for neural network based replacement algorithm. The number of neurons in the hidden layer(s) are reasonably varied, in our experiments, in the range 10 to 20. For T5, even with two hidden layers, Max-Prop learning rule could not outperform the Norm-Cum based results. However, T3's performance is on the same league as T6 (best case) which demonstrates the fact that for the given data, there is a very little change in learning with the increased momentum and learning rate values. The performance of T7 with respect to T6 indicates that very high values of momentum and learning rates do not produce very drastic results as compared to moderate values, but, very low values are indeed detrimental to the performance.

The decrement in the miss ratio with increasing cache size is uniform across different types of BPNNs. Note that the relative performance of T7 in figure 4.3 is worse as compared to figure 4.2. This is due to the fact that the learning of neural networks is dependent on the data file.

4.3 Performance of NEA Algorithm Based on Modular Neural Networks, Radial Basis Functional Networks, and Learning Vector Quantization Neural Networks

Before we discuss the performance of modular neural networks (MNNs), radial basis functional networks (RBFNs), and learning vector quantization networks (LVQs), first let's have a glance at these paradigms. MNNs consists of a group of BPNNs competing to learn different aspects of a problem, and therefore, it can be regarded as a generalization of BPNNs. Figure 4.4 labels these groups as the Local Experts. A Local Expert can also be a MNN for the recursive type of MNNs. There is a Gating Network associated with MNNs that controls the competition and learns to assign different parts of the data space to different networks (Local Experts).

Training of the Local Experts and the Gating Network is achieved using the same rules as for BPNNs. The equations for weight-updation in MNNs are derived from the requirement that the network maximizes the following *Objective function*:

$$J = \mathrm{Ln}\left\{ \sum_{L=1}^{N} g_L \exp\left\{-0.5(d-y_L)^T(d-y_L)\right\}\right\} \qquad (15)$$

Where,

N = number of neurons in the Gating Network output layer.

$g = (g_1,.....,g_N)$ = activation vector of Gating Network output layer.

$d = (d_1,.....,d_p)$ = desired output vector(for the entire network).

p = number of neurons in MNN output layer and also in each of the Local Expert output layers.

$y = (y_1,.....,y_p)$ = output vector(for the entire network).

The normalized output values of the Gating Network are used to weight the output vector from the corresponding Local Experts. The final output vector is the sum of the

weighted output vectors.

In general, a RBFN is any network which makes use of a radially symmetric and radially bounded transfer function in its hidden layer. The MD-RBFN (Moody Darken-RBFN), that we have used in the research, consists of 3 layers: an input layer, a prototype layer, and an output layer (see figure 4.5). The transfer function (T) used by the prototype layer (a generic name used for all intermediate neural network layers is the *hidden layer*) is of the following form:

$$T = \exp\left[(net_j - R^2) / \sigma^2 \right] \qquad (4.10)$$

Where, R = projection radius

σ = width of the radial basis function

The input layer of a RBFN is actually a buffer, and the output layer is effectively an adder due to linear transfer function of the output layer neurons. The learning for the hidden layer neurons is done using one of the clustering algorithms (K-means, SOM, etc.). However, the output layer neurons are trained using the gradient descent algorithm. The advantage of MD-RBFN is that it leads to better decision boundaries than BPNN

when used for classification and decision problems. Also, it trains faster than BPNN. RBFNs are generally used for system modeling, prediction, and classification.

Lastly, LVQ is a classification network. A simple LVQ contains an input layer, a Kohonen layer, and an output layer (see figure 4.6). The input and the output layers are just buffers like most of the other NNs. The Kohonen layer learns and performs the classification. Usually groups of neurons in the Kohonen layer are assigned to particular classes. In the learning/training mode Euclidean distance d is first computed between the weight vector W and the input vector X.

$$d_i = \| W_i - X \| = \left\{ \sum_{j=1}^{N} (w_{ij}-x_j)^2 \right\}^{1/2} \qquad (4.11)$$

Where, N = dimension of input space

After computing the Euclidean distance, the winning PE (Processing Element) or a neuron in the class of the training vector is moved towards the training vector. Whereas, the winning neuron not in the class of the training vector is moved away from the vector. The

weights associated with the winning neurons are adjusted according to the following rules:

$$w^{new} = \begin{cases} w^{old}+\alpha(x-w); & \text{if the winning neuron is in the correct class} \\ w^{old}-\alpha(x-w); & \text{if the winning neuron is not in the correct class} \end{cases}$$

The basic LVQ has various drawbacks and therefore, several variants such as, LVQ1, LVQ2, Extended LVQ, and LVQ without repulsion, have been developed to overcome these shortcomings. The difference between these variants is in how the prototype vectors are learned. In this research, LVQ1 and LVQ2 were cascaded to eliminate the drawbacks of basic LVQ so as to achieve higher performance with the LVQ paradigm.

Figures 4.7 and 4.8 illustrate the relative performance of MNN, RBFN, and LVQ for BI.DATA and TPC.DATA, respectively. Again, each figure is partitioned into three parts (a), (b), and (c), corresponding to the three values of associativity (2, 4, and 8). For most of the cases, observe that LVQ has a relatively better performance than the other two. The difference is particularly significant for higher values

of associativity. With 32 sets and associativity equal to 8, for TPC.DATA trace file, LVQ shows an improvement in the miss ratio of about 5.2% and 5.04% over RBFN and MNN respectively. For BI.DATA, the difference in performance is relatively smaller. For 256 sets and 8-way associativity, the best performance of LVQ for TPC.DATA provides a relative improvement of 5.57% and 5.5% over RBFN and MNN respectively. For a 4-way associativity the best performance occurs for 2048 cache sets (TPC.DATA); here, we see an improvement of 5.4% and 5.54%. With BI.DATA trace file, LVQ's performance remain close to the RBFN and MNN for most of the cache configurations. However, it may be observed that the best performance of LVQ is provided by BI.DATA. Figure 4.7(c) shows that for a cache specified by 512 sets and 8-way set associativity, LVQ's performance is 5.6% and 5.56% better than RBFN and MNN, respectively. Excellent performance of LVQ over RBFN and MNN can be attributed to two factors:

(1) LVQ neural network is derived from Vector Quantization for data compression. This is advantageous because, our proposed interface layer (discussed in section 3.2) essentially compresses the incoming

data/address to determine appropriate classification for the new address referenced by the CPU.

(2) RBFN and MNN suffers from the defect that some of their classifying neurons tend to attract data too often, while others do nothing. This problem is alleviated in LVQ through the *conscience mechanism*. The crux of the conscience mechanism is to support neurons that do nothing. In other words, a neuron that wins very frequently is assigned a guilty conscience and is penalized for having too many successes.

Although LVQ has shown good performance relative to RBFN and MNN paradigms for our given data set and algorithm, yet, it lags behind the results produced by some of the variants of BPNN (see figures 4.2 and 4.3). This problem may be due to several drawbacks associated with the LVQ paradigm such as: improper implementation of conscience mechanism, training of incorrect neurons, and selecting LVQ parameters for establishing incorrect regional boundaries etc.

4.4 Performance of EA Algorithm Based on Probabilistic Neural Networks And General Regression Neural Networks

The probabilistic neural network (PNN), shown in figure 4.9, is actually an implementation of Bayes decision rule. The training data for a PNN is used to develop distribution function, which in turn is used to estimate the likelihood of a feature vector being within the given categories. Optionally, this can be combined with the a priori probability of each category. PNN uses Parzen windows to estimate the probability density functions (pdf) related to each class. The Parzen estimator used in a PNN is of the form:

$$f_k(x) = \frac{1}{2\,\pi^{Y/2}\sigma^{Y}} \frac{1}{N_k} \sum_{j=1}^{N_k} \exp[-((x-x_{kj})^T(x-x_{kj}))/2\sigma^2 \quad (4.12)$$

Where, k = class number/category number.

x_{kj} = j^{th} training sample (tag) in class k.

N_k = no. of training samples in category k.

$\sigma = \sigma(N_k)$ = smoothing parameter.

Y = dimension of input vector x.

The input layer of a PNN shown in figure 4.9 is a buffer. The next layer is usually a Normalizing layer that normalizes the incoming data vector. The Pattern slab/layer is the layer that is used for representing the training samples. Since, number of data items are very large, therefore, the number of neurons in the Pattern layer needs to be fixed to an appropriate value. The value was chosen to be A (Associativity) for the present case. The Pattern layer develops Parzen window kernels, which are eventually added up by the Summation layer neurons. The last layer of PNN is called the Win Class slab/competitive layer. The purpose of the Win Class slab is to identify the winning class/neuron for the given input data. The main advantage of PNN is that it can be effectively used with sparse data.

Next, general regression neural network (GRNN) is a general purpose neural network paradigm, primarily used for system modeling and prediction. The main advantage of GRNNs is that they can be effectively used with sparse data. Also, they are capable of handling non-stationary data. GRNNs can be regarded as the generalization of probabilistic neural networks. GRNNs use the same Parzen estimator as the PNNs. The main

disadvantage of such neural networks is that they have a slow response time due to memory intensive activities. A simple GRNN consists of an input layer, a pattern layer, summation and division layer, and an output layer. In figure 4.10 connections are shown for only one pattern unit; all other pattern units have similar connections and weights. Again, the input layer acts as a buffer. The pattern layer forms the Gaussian Kernels. These kernels are used in the summation and division layers to implement the desired statistical equations.

Now, let's look at the comparative performance of a PNN and a GRNN, as illustrated via figures 4.11 and 4.12. Performance of both the estimating type of neural networks appear to be very much identical. This is understandable since both, PNNs and GRNNs, rely on developing probability density functions using Parzen estimator. Parzen estimation is a non-parametric estimation technique for approximating density functions. Such functions are constructed from many simple parametric probability density functions (pdfs). These simple pdfs, which are typically Gaussian, are known as Parzen windows. An important difference between a PNN and a GRNN is that the PNN is specifically

tailored to solve classification problems, and it is mostly preferred over GRNN for such problems. This explains why, for most of the cache configurations, PNN outperforms GRNN. The problem of recognizing the difference between live and dead lines in a cache is viewed as a kind of classification problem by PNN. Again, as always, the performance of PNN shows a marked improvement over GRNN only for higher values of associativity. Figure 4.12 illustrates that the best performance of PNN with respect to GRNN occurs for a cache having 2048 sets and 8-way associativity (TPC.DATA). The best relative improvement was found to be 2.09%. But, for the similar cache configurations, the improvement was found to be only 0.59% with BI.DATA trace file. For 1024 sets and 8-way associativity the relative improvement becomes comparable, for the two trace files TPC.DATA and BI.DATA, showing values of 1.68% and 1.87% respectively. All other cache configurations yield a difference of less than 1% in the miss ratio values produced by PNN and GRNN. Lastly, note that PNN always outperforms GRNN for the given data set/trace files.

4.5 Miss Ratio Analysis for Optimal (OPT), First In First Out (FIFO), Most Recently Used (MRU), Least Recently Used (LRU), NEA (Best Paradigm), and EA (Best Paradigm) Algorithms

This section presents a comparative performance of the popular conventional algorithms and the best performing neural network paradigms used with the NEA and EA algorithms. Looking at the results in figures 4.13 and 4.14, we observe that the conventional algorithms follow a well-defined performance hierarchy rule, that is, the performance of these algorithms were found to be in the following general order (from high to low): OPT, LRU, FIFO, and MRU. OPT has the best performance since it has perfect knowledge of the future references. The other three conventional algorithms use past history of references in different ways to determine the line to be replaced on a cache miss. LRU and MRU are usage-based algorithms, that is, they consult the usage record of the line when performing line replacements. FIFO is a non-usage based algorithm since it only looks at the arrival record of a line. The arrival record only contains a narrow perspective of the history of references because, the oldest line in a cache set may

very well be the most frequently referenced line, and hence, its replacement could result in future cache misses. This suggests that usage-based algorithms should exhibit a relatively better performance than its counterpart. However, figures 4.13 and 4.14 illustrate that MRU consistently exhibits lower performance than FIFO although it is a usage-based algorithm. This is because, MRU does not allow new cache arrivals to stay resident for a sufficient period of time. As a result, the incoming lines are usually not given ample chances to reveal their true behavior. Therefore, MRU is not able to take into account the correct usage of a line.

The preceding discussion on simulation results suggests two important factors that determine the relative performance of practical/non-lookahead cache replacement algorithms, they are: Type of information extracted from the history of references, and the length of history consulted. There are many other secondary factors that dictate the relative behavior of replacement algorithms such as: the characteristics of trace files, size of caches, and cache parameters.

Comparing the performance of conventional schemes with the neural network based approaches, we observe from figure 4.13 and 4.14 that our NN based algorithms were successful in bridging the large gap between LRU and OPT. The NN paradigms selected for comparison with the conventional algorithms were BPNN and PNN. For the peak category, BPNN provides a significant performance improvement of 16.4711% in the miss ratio values over the LRU algorithm (Associativity=2, No. of Sets=2048, Trace file= TPC.DATA). The average performance improvement is not significant because of the large variance in BPNN's performance as depicted by figures 4.2 and 4.3. PNN also performs well and in most of the cache organizations considered in the simulation studies, its performance was found to be better than the LRU (see figures 4.13 and 4.14). We expect that longer trace files for a variety of applications and systems software can provide us with even better results.

4.6 Conclusions

This chapter began with an introduction of BI.DATA and TPC.DATA trace files. Cache simulator was then introduced. Performance of NEA and EA algorithms was presented, discussed, and analyzed for several neural network paradigms. In this chapter, six different types of neural networks were studied and evaluated for the line replacement problem in caches. Neural network paradigms considered were: BPNN, MNN, RBFN, LVQ, PNN, and GRNN. The performance of BPNN was found to be excellent for the problem at hand. Backpropagation neural network based replacement scheme showed a significant improvement of approximately 16.47% in the miss ratio values over the most popular conventional line replacement algorithm (LRU).

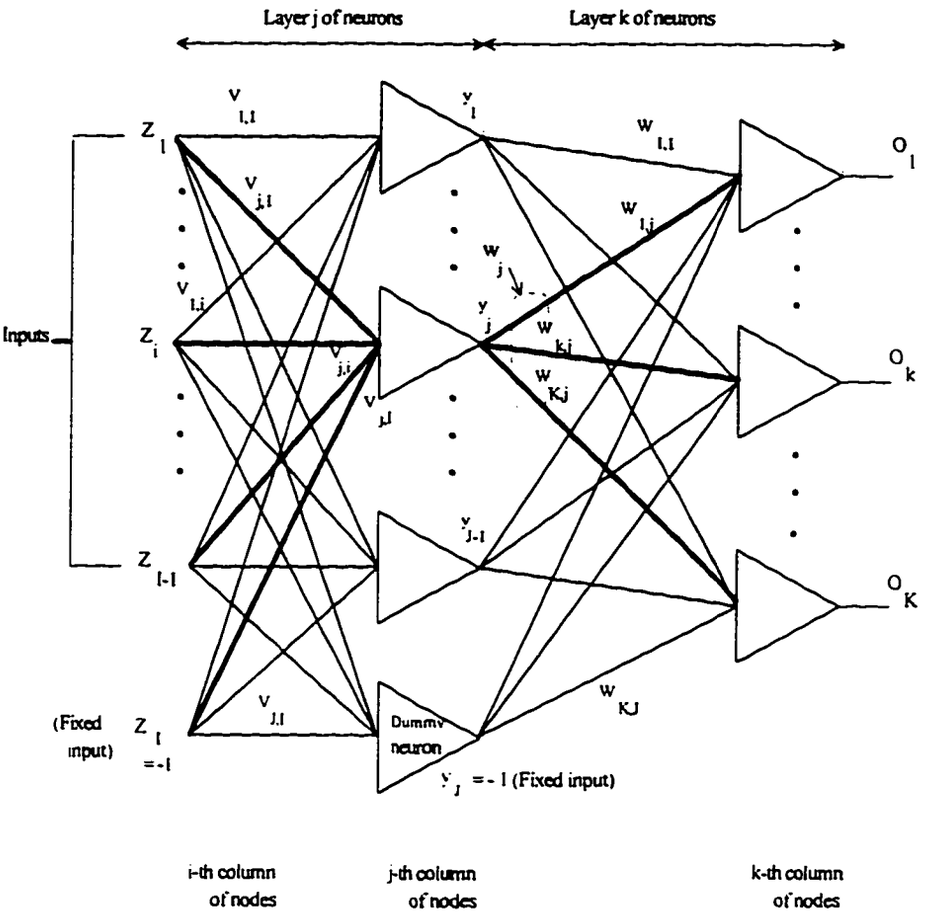


FIGURE 4.1: BACKPROPAGATION NEURAL NETWORK (BPNN).

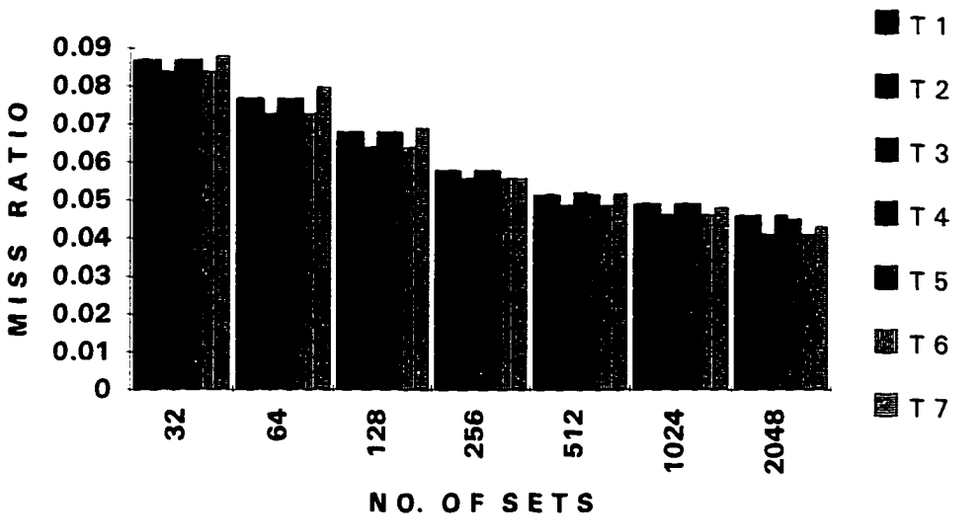


FIGURE 4.2(a): MISS RATIO vs. CACHE SIZE FOR DIFFERENT VARIANTS OF BPNNs WITH BI.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 2

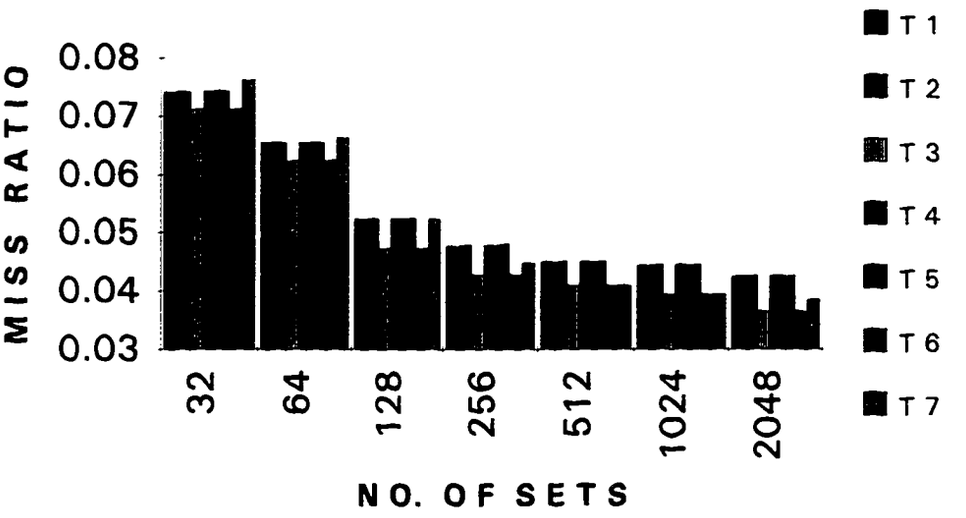


FIGURE 4.2(b): MISS RATIO vs. CACHE SIZE FOR DIFFERENT VARIANTS OF BPNNs WITH BI.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 4

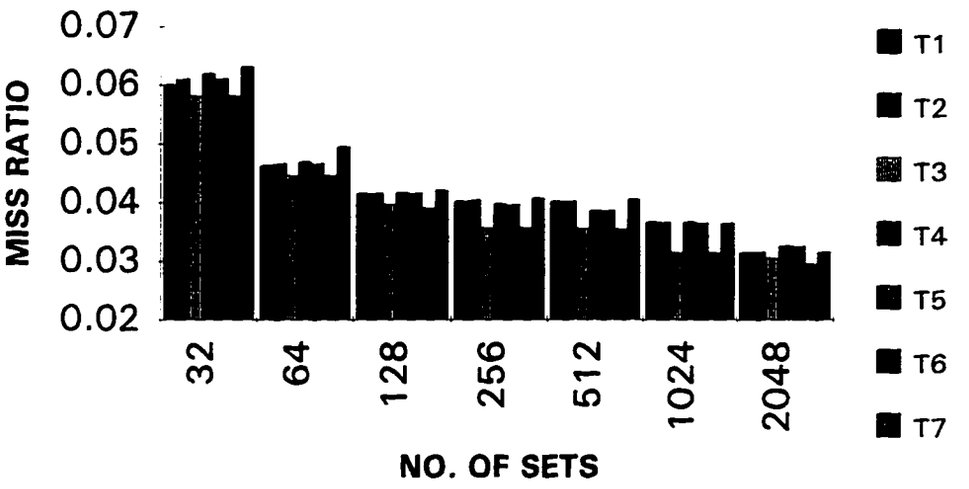


FIGURE 4.2(c): MISS RATIO vs. CACHE SIZE FOR DIFFERENT VARIANTS OF BPNNs WITH BI.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 8

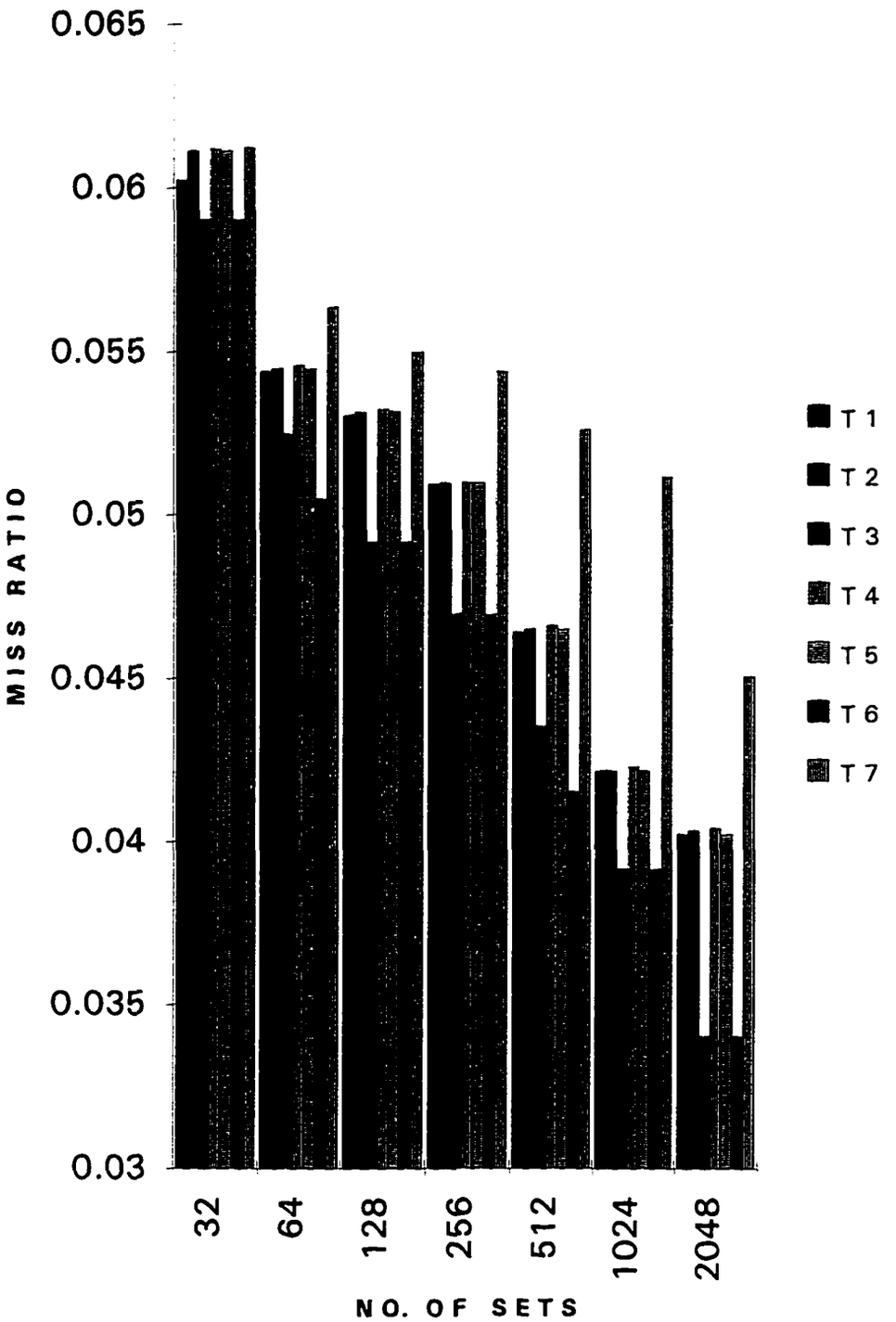


FIGURE 4.3(a): MISS RATIO vs. CACHE SIZE FOR DIFFERENT VARIANTS OF BPNNs WITH TPC.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 2

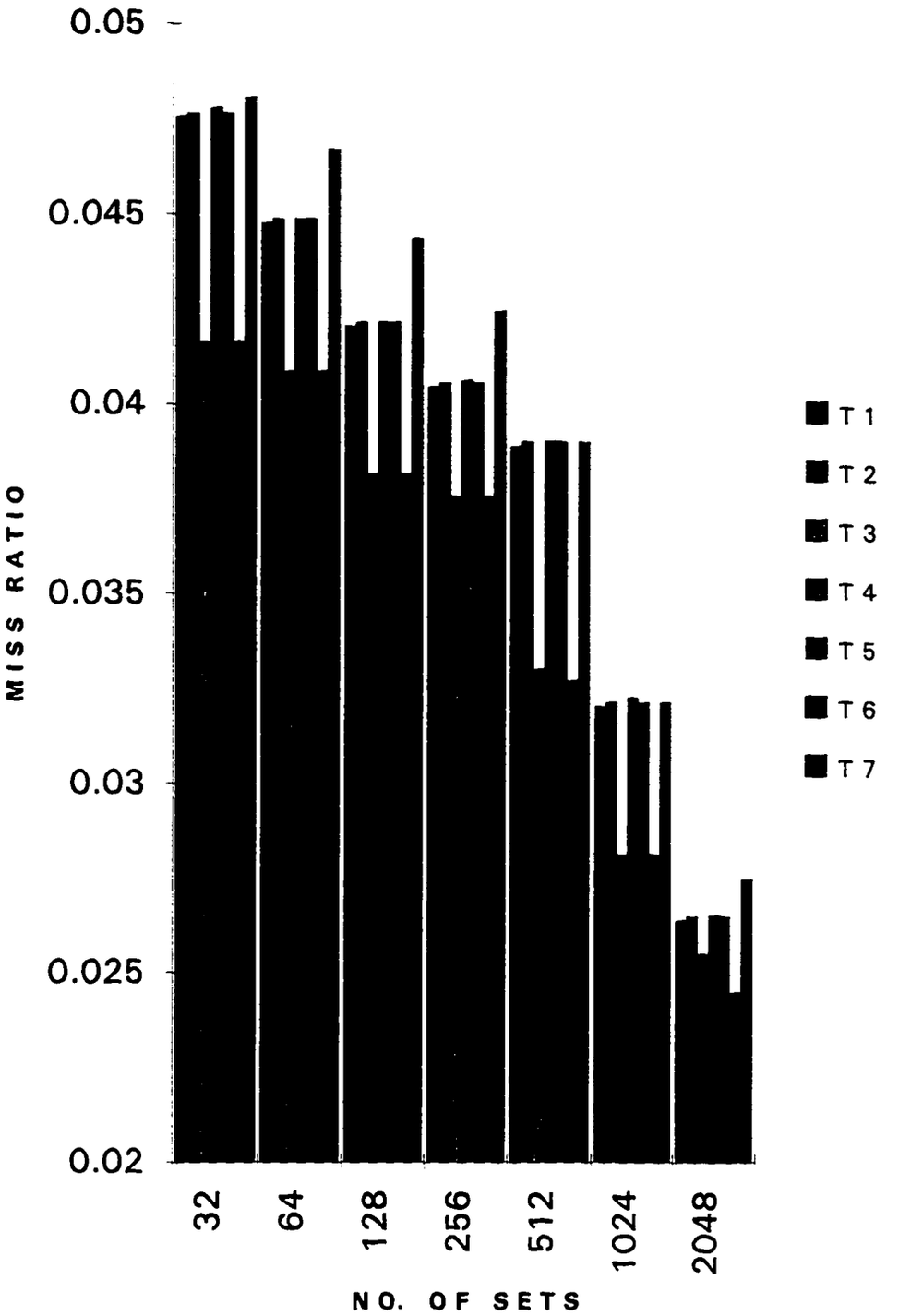


FIGURE 4.3(b): MISS RATIO vs. CACHE SIZE FOR DIFFERENT VARIANTS OF BPNNs WITH TPC.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 4

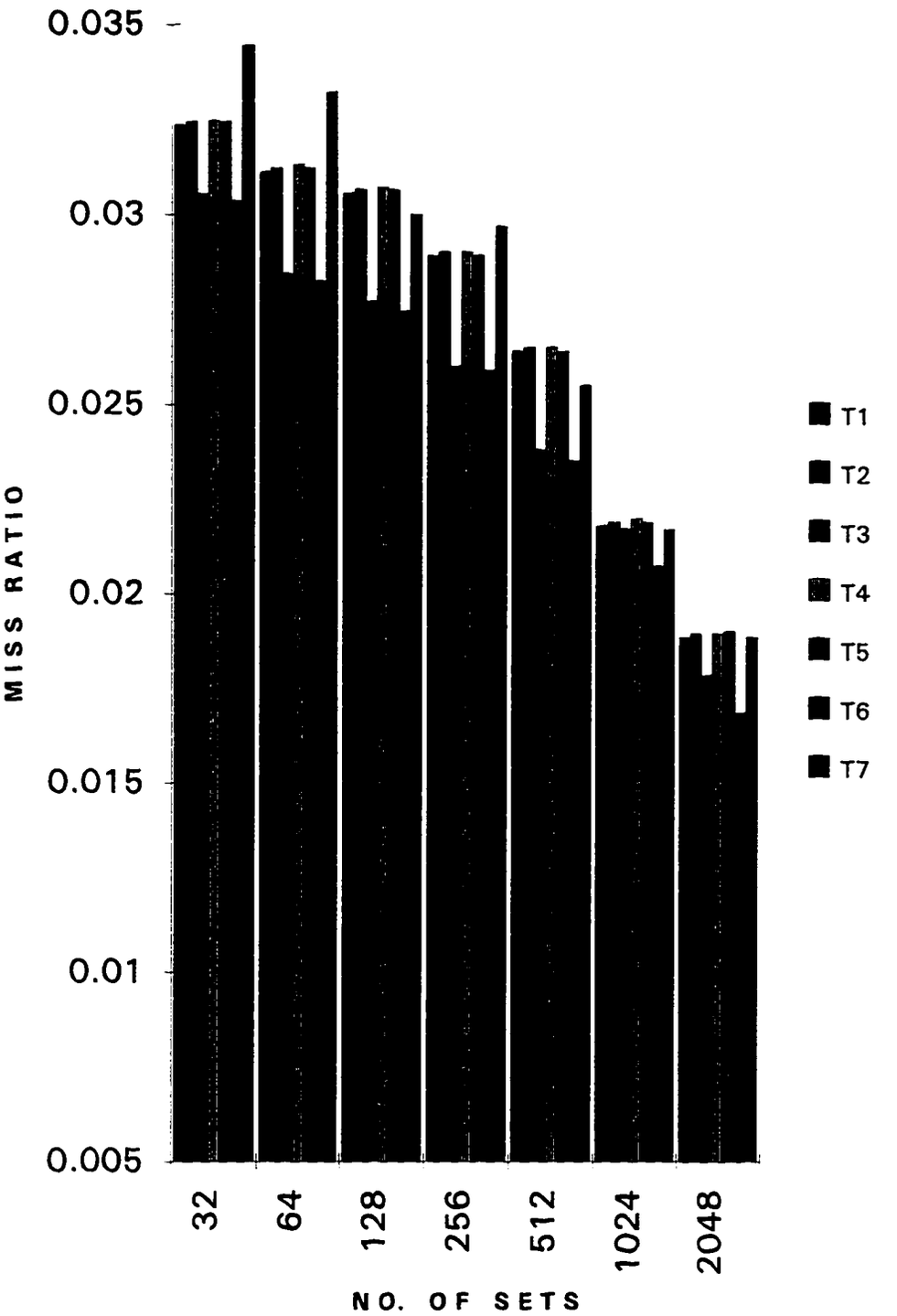


FIGURE 4.3(c): MISS RATIO vs. CACHE SIZE FOR DIFFERENT VARIANTS OF BPNNs WITH TPC.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 8

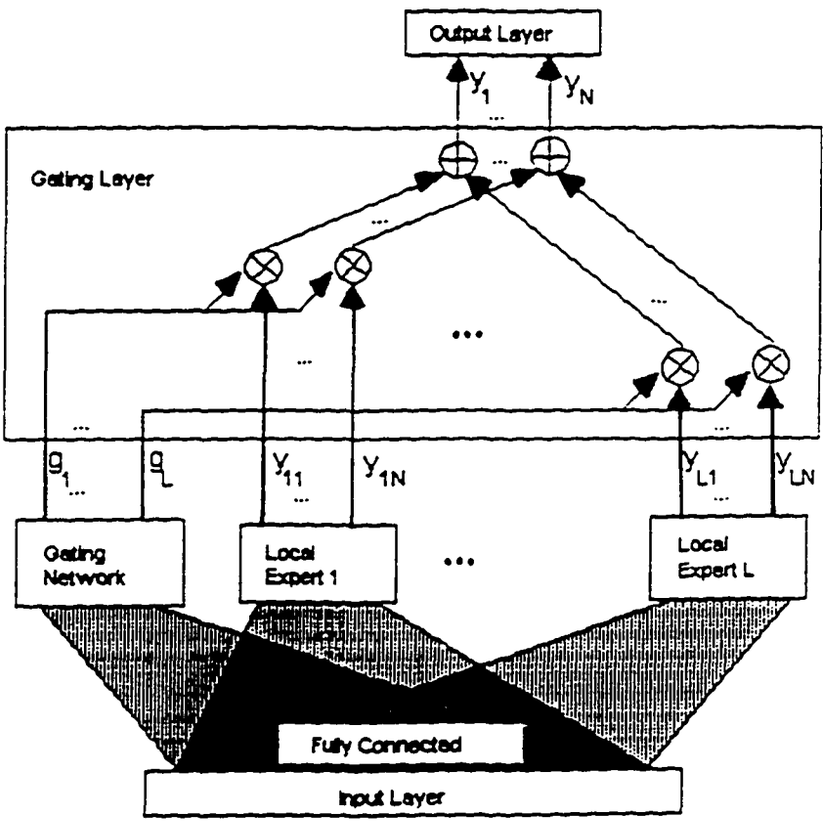


FIGURE 4.4: MODULAR NEURAL NETWORK (MNN)

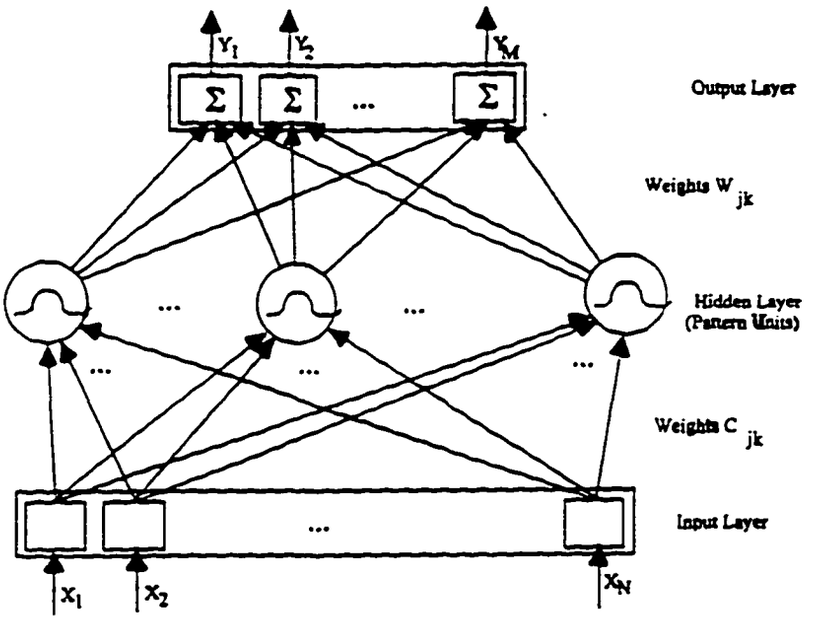


FIGURE 4.5: RADIAL BASIS FUNCTION NETWORK (RBFN)

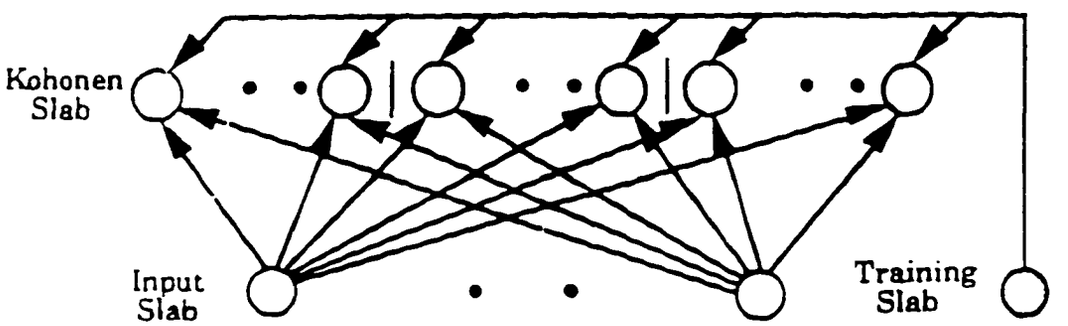


FIGURE 4.6: LEARNING VECTOR QUANTIZATION NEURAL NETWORK (LVQ).

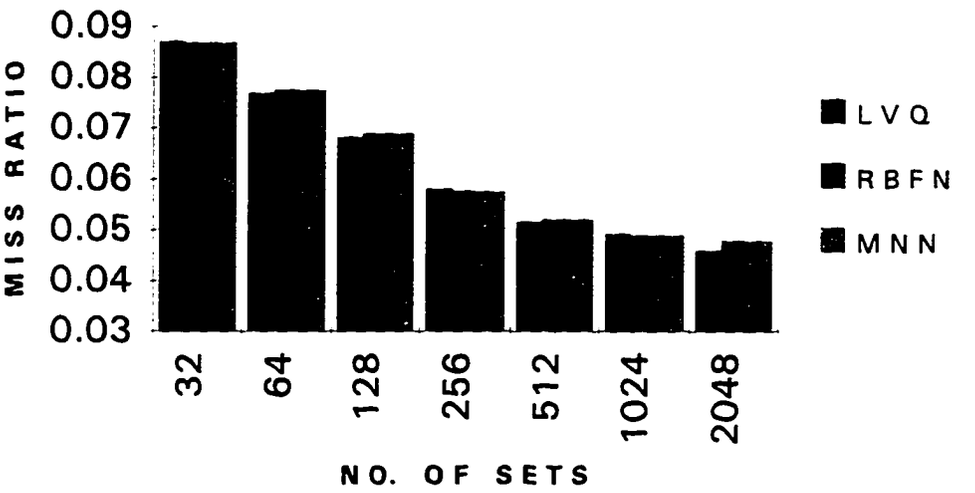


FIGURE 4.7(a): MISS RATIO vs. CACHE SIZE FOR LVQ, RBFN, AND MNN NEURAL NETWORK PARADIGMS WITH BI.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 2

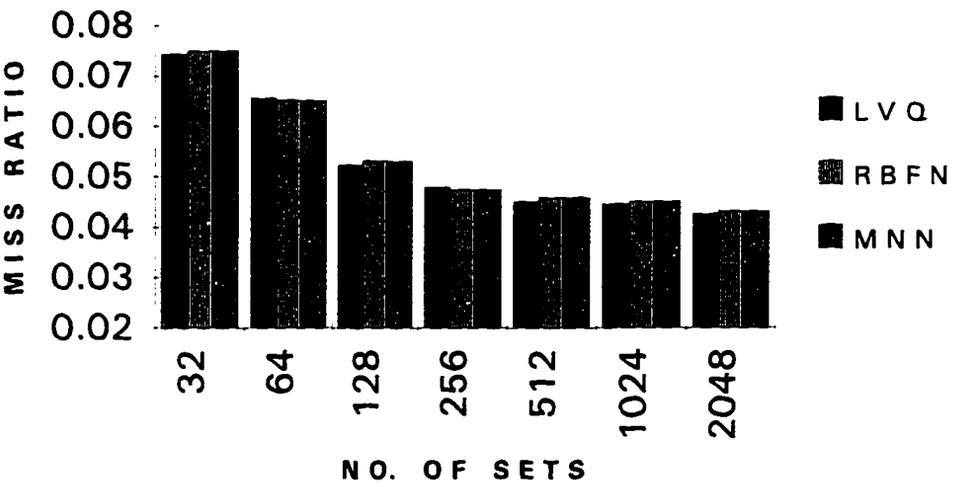


FIGURE 4.7(b): MISS RATIO vs. CACHE SIZE FOR LVQ, RBFN, AND MNN NEURAL NETWORK PARADIGMS WITH BI.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 4

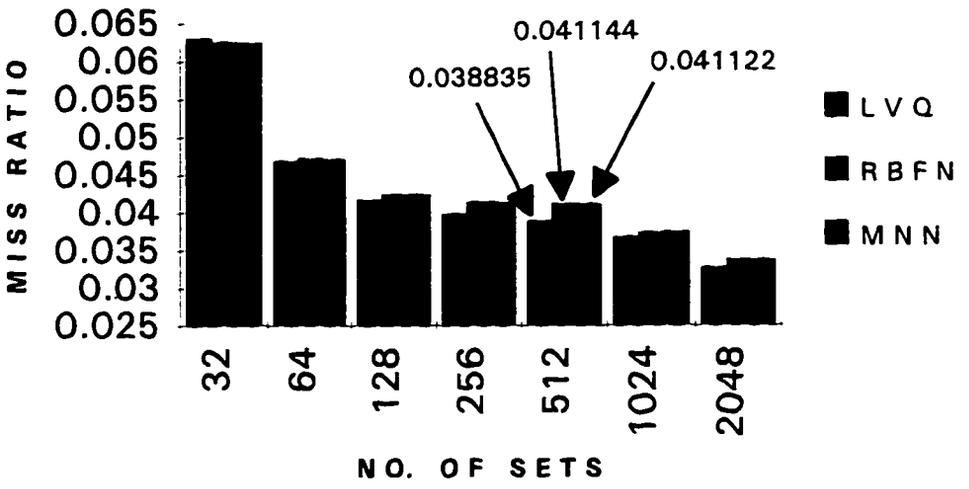


FIGURE 4.7(c): MISS RATIO vs. CACHE SIZE FOR LVQ, RBFN, AND MNN NEURAL NETWORK PARADIGMS WITH BI.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 8

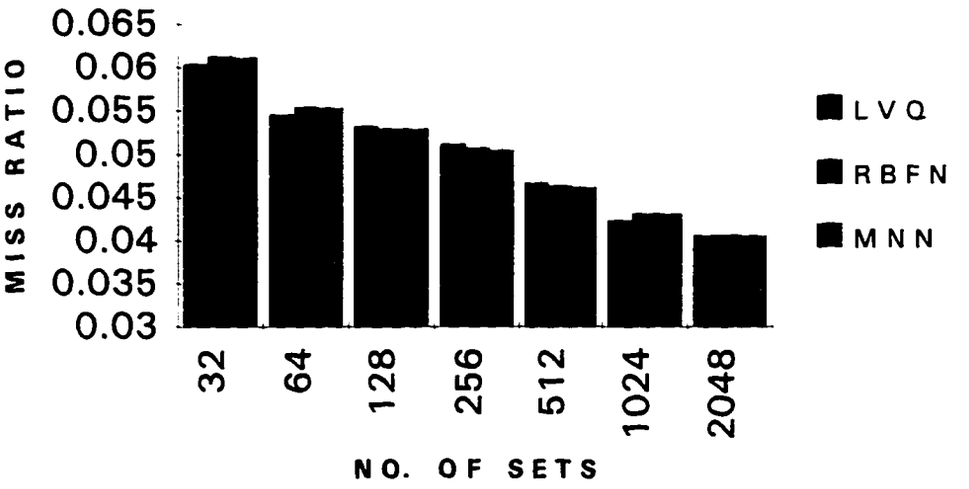


FIGURE 4.8(a): MISS RATIO vs. CACHE SIZE FOR LVQ, RBFN, AND MNN NEURAL NETWORK PARADIGMS WITH TPC.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 2

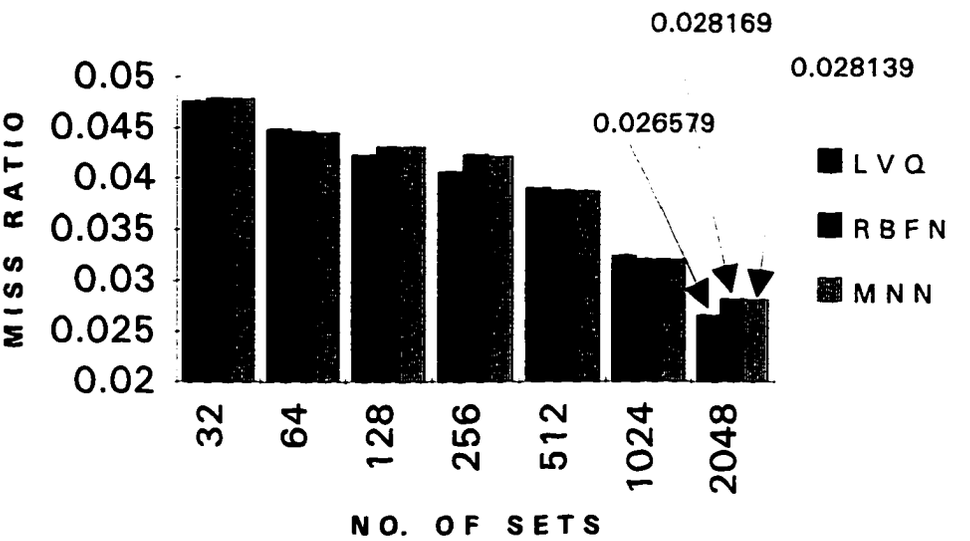


FIGURE 4.8(b): MISS RATIO vs. CACHE SIZE FOR LVQ, RBFN, AND MNN NEURAL NETWORK PARADIGMS WITH TPC.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 4

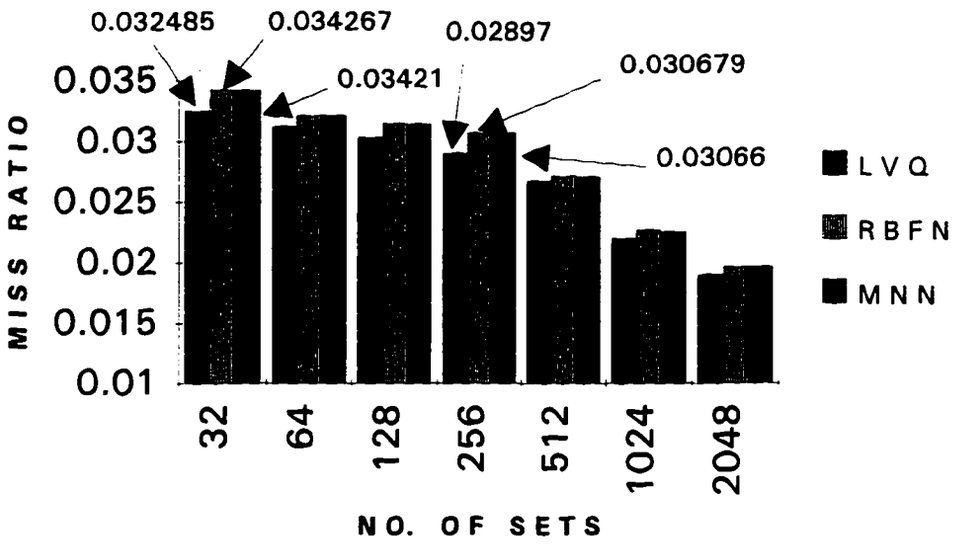


FIGURE 4.8(c): MISS RATIO vs. CACHE SIZE FOR LVQ, RBFN, AND MNN NEURAL NETWORK PARADIGMS WITH TPC.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 8

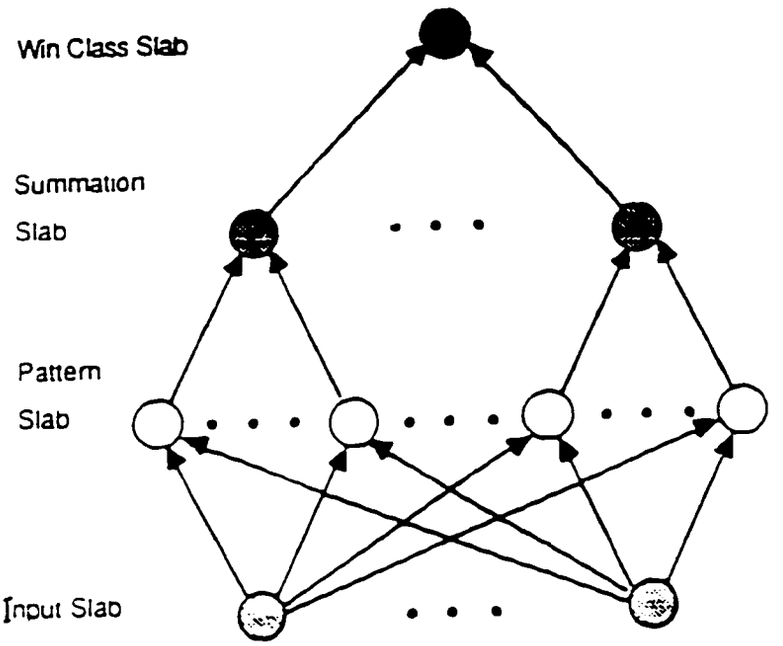


FIGURE 4.9: PROBABILISTIC NEURAL NETWORK (PNN)

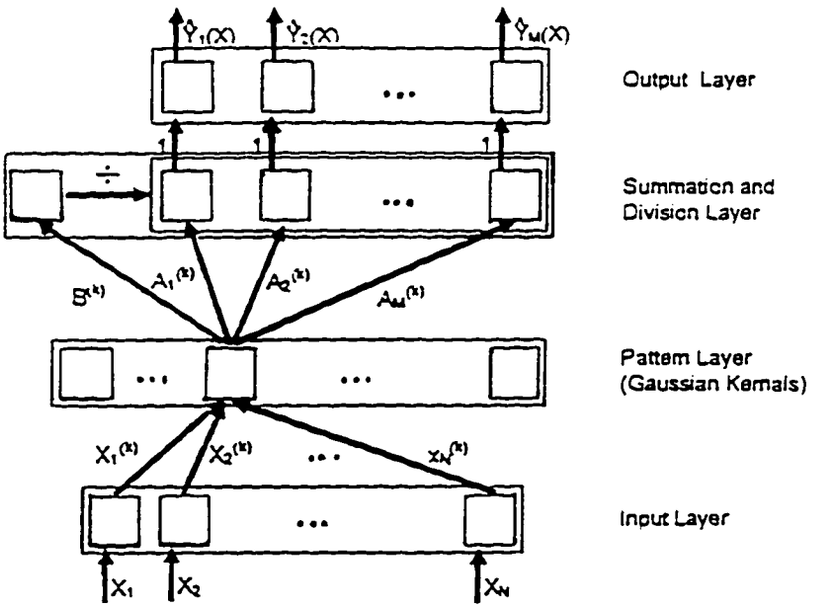


FIGURE 4.10: GENERAL REGRESSION NEURAL NETWORK (GRNN).

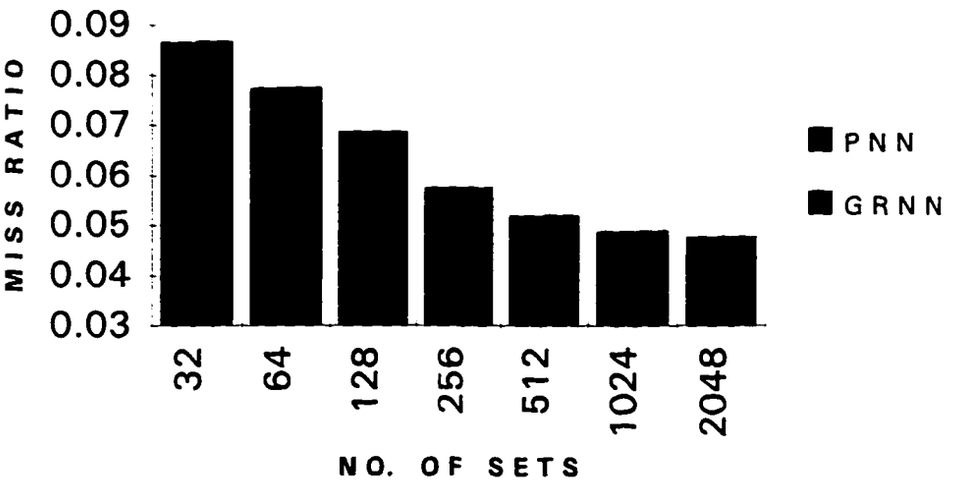


FIGURE 4.11(a): MISS RATIO vs. CACHE SIZE FOR PNN AND GRNN NEURAL NETWORK PARADIGMS WITH BI.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 2

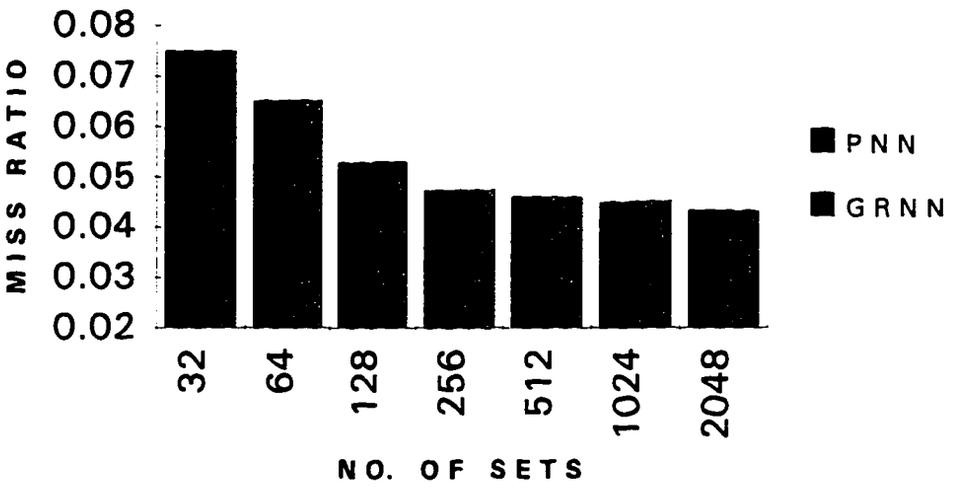


FIGURE 4.11(b): MISS RATIO vs. CACHE SIZE FOR PNN AND GRNN NEURAL NETWORK PARADIGMS WITH BI.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 4

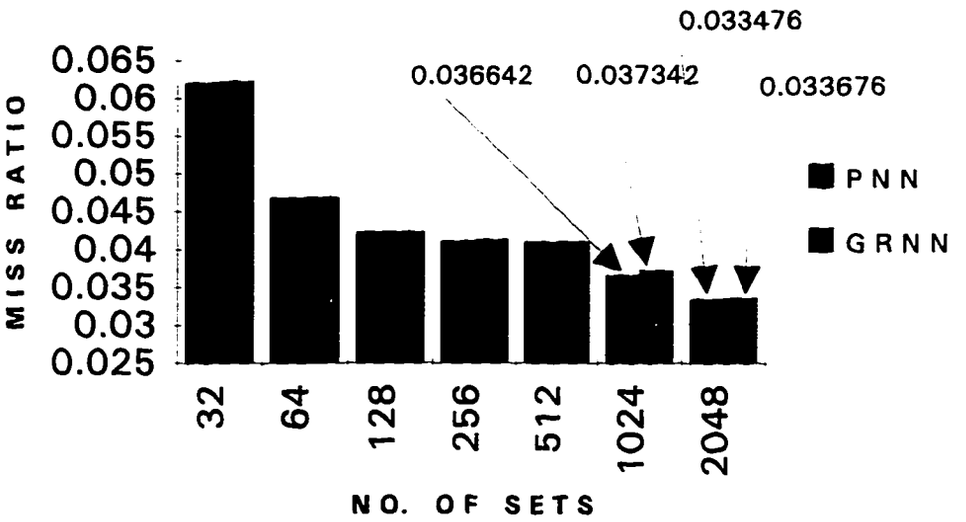


FIGURE 4.11(c): MISS RATIO vs. CACHE SIZE FOR PNN AND GRNN NEURAL NETWORK PARADIGMS WITH BI.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 8

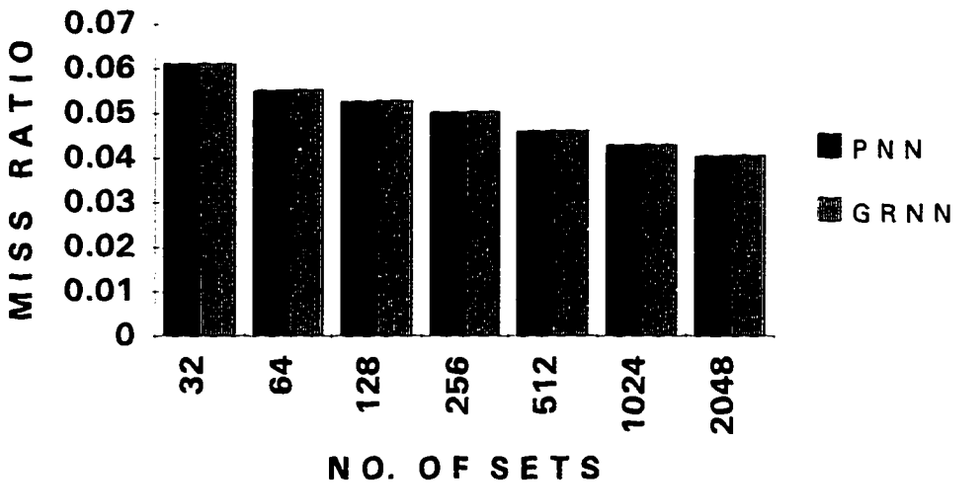


FIGURE 4.12(a): MISS RATIO vs. CACHE SIZE FOR PNN AND GRNN NEURAL NETWORK PARADIGMS WITH TPC.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 2

FIGURE 4.12(b): MISS RATIO vs. CACHE SIZE FOR PNN AND GRNN NEURAL NETWORK PARADIGMS WITH TPC.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 4

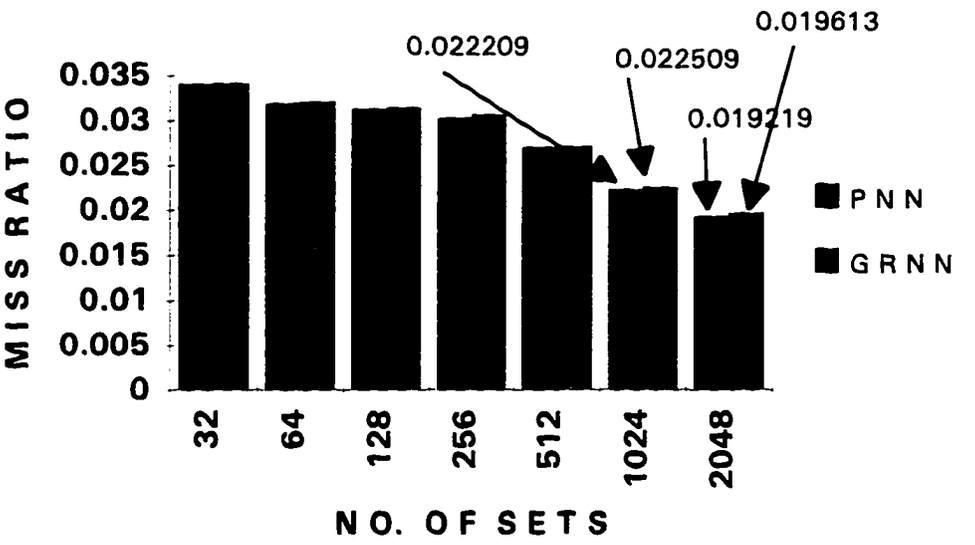


FIGURE 4.12(c): MISS RATIO vs. CACHE SIZE FOR PNN AND GRNN NEURAL NETWORK PARADIGMS WITH TPC.DATA AS THE TRACE FILE AND ASSOCIATIVITY = 8

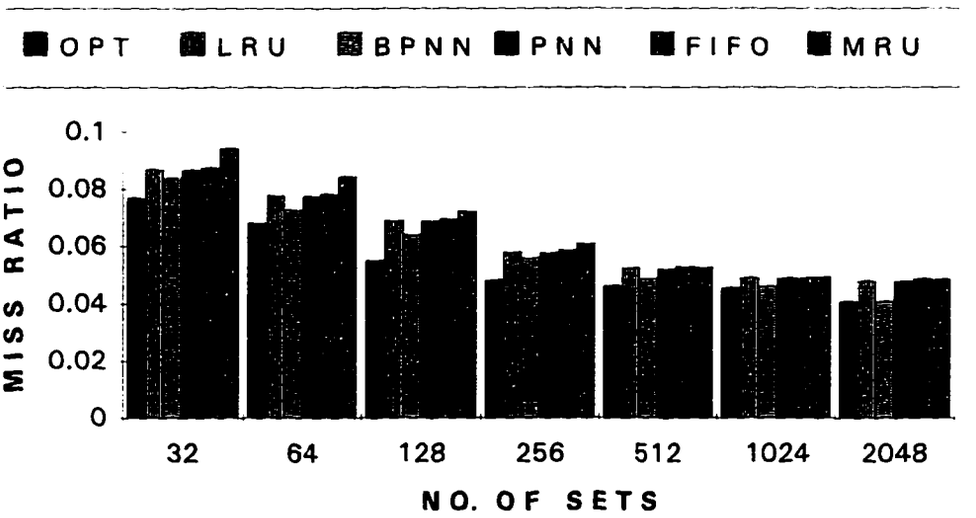


FIGURE 4.13(a): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, BPNN, PNN, FIFO, AND MRU WITH BI.DATA AS THE TRACE FILE (ASSOCIATIVITY = 2)

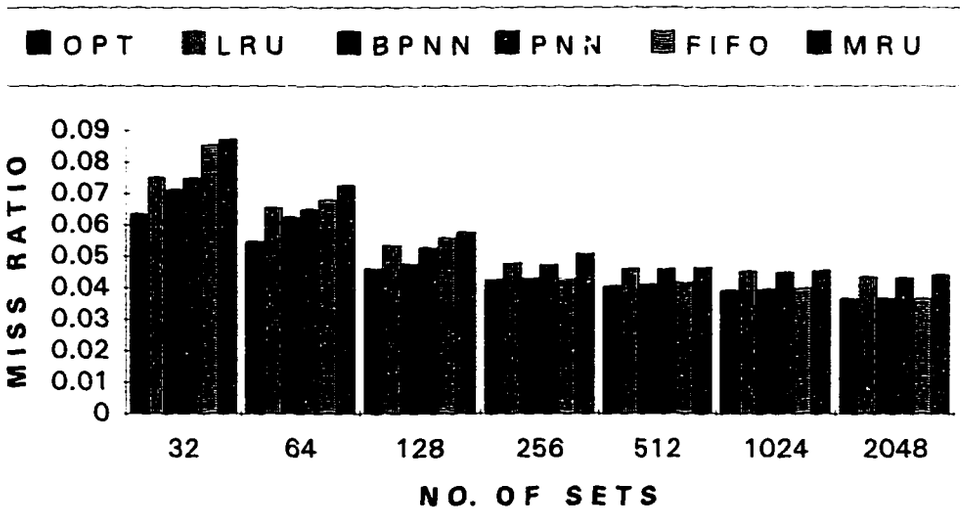


FIGURE 4.13(b): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, BPNN, PNN, FIFO, AND MRU WITH BI.DATA AS THE TRACE FILE (ASSOCIATIVITY = 4)

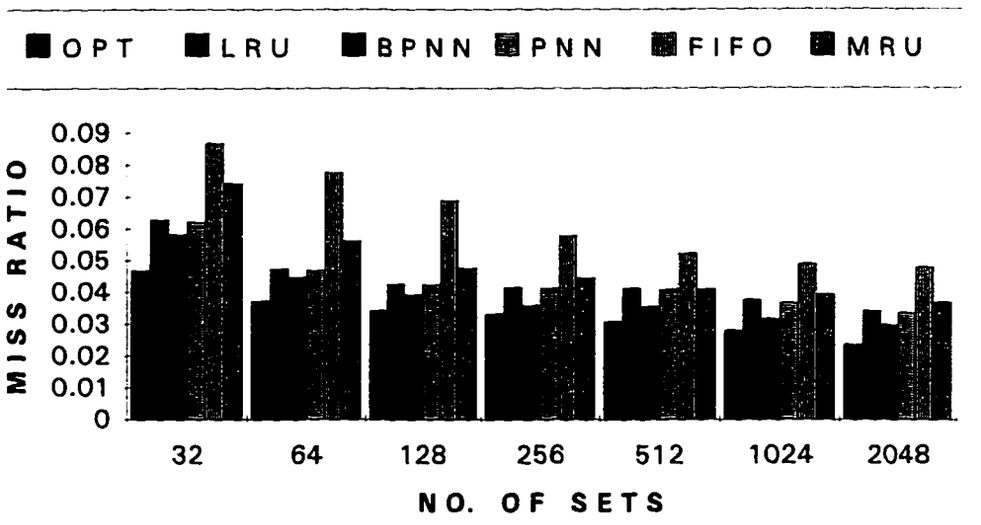


FIGURE 4.13(c): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, BPNN, PNN, FIFO, AND MRU WITH BI.DATA AS THE TRACE FILE (ASSOCIATIVITY = 8)

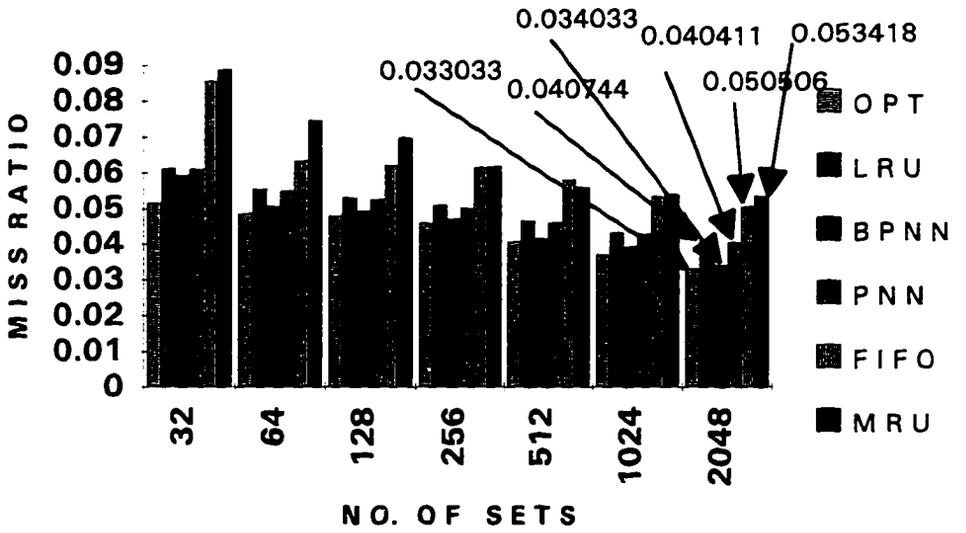


FIGURE 4.14(a): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, BPNN, PNN, FIFO, AND MRU WITH TPC.DATA AS THE TRACE FILE (ASSOCIATIVITY = 2)

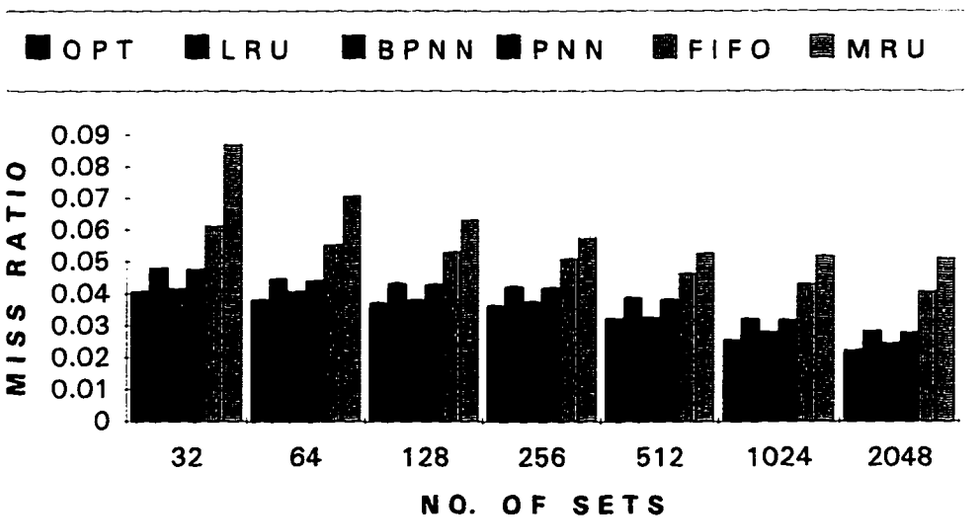


FIGURE 4.14(b): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, BPNN, PNN, FIFO, AND MRU WITH TPC.DATA AS THE TRACE FILE (ASSOCIATIVITY = 4)

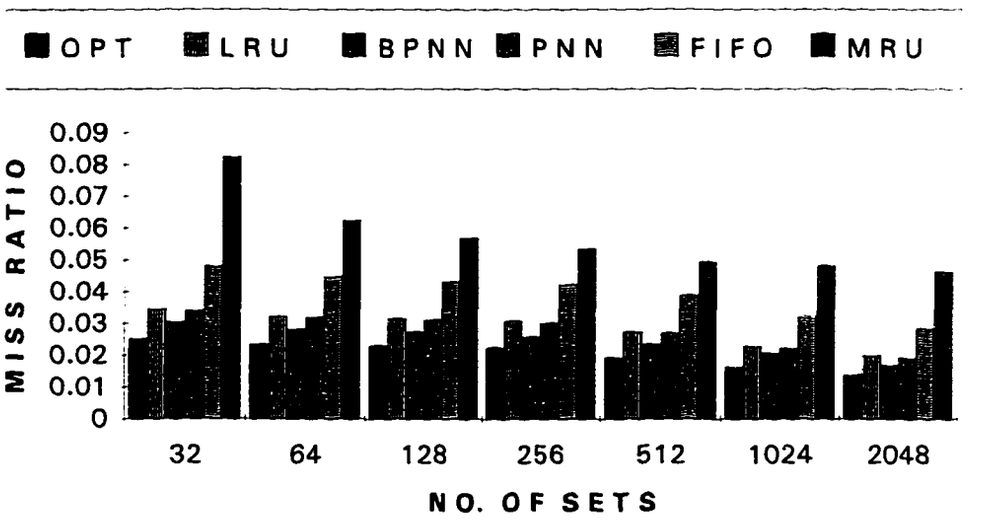


FIGURE 4.14(c): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, BPNN, PNN, FIFO, AND MRU WITH TPC.DATA AS THE TRACE FILE (ASSOCIATIVITY = 8)

CHAPTER 5

CONSTRUCTION AND ANALYSIS OF A GENERALIZED CACHE REPLACEMENT ALGORITHM

5.1 Development of a Generalized Cache Replacement Algorithm for Estimating and Non-Estimating Neural Networks [GA]

In the preceding chapters, most of the questions that were raised in the beginning of this thesis were answered. We have learned that it is possible to develop algorithms which can outperform LRU. By reformulating the line replacement problem in caches as being one related to the recognition of live and dead lines, we were able to use neural network techniques as a powerful optimizing/predicting tool. We learned how to develop cache replacement algorithms suitable for different classes of neural networks. We investigated the relative performance of different neural network paradigms, for a practical data set, in order to identify neural network(s) suitable for the problem at hand. Extensive simulations were performed and the results for the proposed neural network based algorithms were compared

with the results from popular conventional algorithms. The implementation issue was also addressed.

In this section, a new cache replacement algorithm suitable for both, estimating and non-estimating, categories of neural networks is formulated and investigated. In particular, we are looking for an algorithm that achieves near-optimal performance.

A near-optimal replacement algorithm can be developed by using LRU as the basis of its development. In section 2.2, it was observed that LRU and OPT stacks were very much alike. This suggests that if an algorithm is developed so that it slightly offsets the replacement decisions made by LRU, resulting in the flushing of additional dead lines from the LRU stack, it may be possible to achieve near-optimal performance with the algorithm. Such an algorithm would be considered a generalized algorithm if the neural networks used for complementing LRU decisions belong to any of the aforementioned neural network types. Thus, for the generalized neural network based algorithm, the NNs should not be trained in a manner similar to the one discussed in sections 3.2 and 3.3. In this case, NNs

could not be used to predict or identify the cache lines to be replaced from within a set, since this would be the job of the algorithm that maintains the pseudo-LRU stack. Moreover, for non-estimating NNs used with such an algorithm, there is a need to investigate new and alternate method(s) to develop the probability density function of addresses.

In an attempt to develop a generalized neural network based cache replacement algorithm that behaves as a pseudo-LRU, the concept of *Shadow Directory* proposed by Pomerene et al. [25] is used with appropriate modifications in this research. Pomerene's concept of Shadow Directory was also adopted by T.R. Puzak to develop a set of near-optimal cache replacement algorithms [31]. However, even for the primitive trace files, Puzak's algorithms were not able to achieve significant performance improvement over the LRU.

According to Pomerene, a shadow directory is actually an extension to the cache's real directory. It identifies only those lines that would be contained in a larger physical cache. Therefore, no physical cache memory is associated with the shadow directory. Figure

5.1 shows an A-way cache organization with N-sets along with a shadow directory. An ordinary cache is divided into two parts: Main directory and the data area containing cache lines; this constitutes the left portion of figure 5.1. The rightmost part is the shadow directory which contains only the tags discarded from the main directory. The shadow based cache of Pomerene with associativity *A* is managed as a 2A-way cache. A hit in the shadow directory portion of the cache results in the directory information only. When a new line is brought into the main cache, one of *A* lines is discarded from a set to which the item maps. The tag for the discarded line is then entered in the shadow directory at the most recently discarded position of the corresponding set. All the other tags in the shadow directory are consequently moved down one position and the least recently discarded tag (discarded tag means the tag that corresponds to a discarded line from the main cache) is deleted.

The main purpose behind the concept of shadow based cache is to identify two kinds of misses: transient misses and shadow misses. A transient miss is one in which a referenced line is not found in either the main

cache directory or the shadow directory. Whereas, in a shadow miss the referenced line is only absent from the main cache directory. This implies that a shadow miss refers to a line that was used in the distant past and is being used again. Since, there is a probability that the same reference characteristics would be repeated in future, Pomerene et al. labeled each incoming line to a cache as being either a shadow or a transient. The shadow/transient status of a line was then used by the cache controller to make replacement decisions. The main idea in Pomerene's approach is to retain lines with shadow status in favor of the lines that have transient status. This enabled the algorithm to flush out transients from cache more quickly than done by LRU algorithm. However, there are several problems associated with Pomerene's approach:

(1) Performance of Pomerene's approach was reported to be low for small sized shadow directory. The performance was found to be some sort of rising function of the increasing shadow directory size.

(2) The shadow directory based scheme is also rigid and crude in determining the shadow/transient status of a

line. This is because, it does not take into consideration certain related characterizing parameters such as: line access frequency, how recently lines have been accessed or referenced etc.

The key to our GA algorithm is to use neural networks (NNs) as a *kind of Shadow Directory*. By definition, the shadow lines are the ones which have been seen by the cache recently. Therefore, according to the spatial and temporal locality of caches, there is a high probability that the shadow lines are going to be referenced soon. Hence, the job of our algorithm is to look for the shadow lines, as identified by the NNs, and provide them with a preferential treatment. Transient lines are given chances to improve their status. The number of such chances, as provided by the current version of GA algorithm, is one-half the stack depth. Where, the stack depth is defined as being equal to the associativity of the cache. If the transient lines do not become active during the allocated amount of time, they are labelled as potentially inactive (dead), and consequently removed from the cache. First half of the cache stack (containing relatively recent arrivals) is managed by the LRU replacement strategy. As for the second half

(named *priority depth/width region*), the LRU scheme is modified to take into consideration the preferential status of certain cache lines (shadow lines). To implement the above mentioned strategy, we need two extra bits, a β and a γ bit, attached to the real cache directory. Directory entry with γ=1 represent those lines that are to receive preferential status, while the lines with β=1 identify shadow lines (a shadow line may lose its preferential status if it is not rereferenced within a stack cycle). The basic idea is to retain the lines, chosen for replacement, that are classified as having preferential status. Once a line has lost its preferential status (γ=0) it becomes a candidate for replacement (i.e. classified as a shadow line on a probation), and does not bias the LRU replacement rules any more. For the GA algorithm, we have chosen to limit the pseudo-LRU stack operations to within one-half the stack depth. This was done in order to prevent GA algorithm from deleting relatively newer lines very quickly. Hence, the mistakes embedded in the MRU strategy are avoided.

The term " *a kind of Shadow Directory* " for the directory implemented via neural network was used

earlier. This was done due to the fact that NNs cannot actually store all the tag and set fields of the memory references. All it does is that it attempts to *learn* and identify references belonging to particular cache set(s). The learning not only stores a few prototype reference patterns in neural networks, but, it also embed other important information such as: line access frequency, how recently lines have been accessed or referenced etc. Therefore, the hint provided by neural networks regarding transient/shadow status of lines is definitely more informative as compared to the one provided by the shadow directory.

The GA algorithm is described by the following steps accompanied with the corresponding pseudo code:

Step 1: Initialize all weights in the neural network (NN) to small values very close to zero.

Pseudo code:

```
Random_Weights (V,W)
{ Repeat { for j ← 0 to (size_of_hidden_layer[m]-1)
           for k ← 0 to (size_of_layer[m-1]-1)
                V[j][k][m] ← small_random_number()
```

```
        }
Until (all hidden layers are done)

   for j ← 0 to (size_of_output_layer-1)
       for k ← 0 to (size_of_last_hidden_layer-1)
                W[j][k] ← small_random_number()

comment 1: V=Hidden layer weight matrix
comment 2: W=Output layer weight matrix
}
```

Step 2: Partition each incoming memory address into tag, word, and set fields for the set associative cache.

```
Pseudo code:
reference ← Get_Reference (Trace File)
Extract_Fields ()
{ line_bits ← Convert_Into_Bits (line_size)
  set_bits ← Convert_Into_Bits(max_no_of_sets)
  tag_bits ← (reference_bits) - (line_bits + set_bits)
  word_number ← Get_Field (reference, line_bits)
```

```
  set_number ← Get_Field (reference, set_bits)
  tag_field ← Get_Field (reference, tag_bits)
}
```

Step 3: The tag and the set fields of a memory reference are jointly applied to the input layer of a neural network (see figure 5.2). The number of neurons in the output layer of the neural network should have one-to-one correspondence with each set of the cache memory. The response of the network's output neuron for a memory reference, corresponding to a particular set, is compared to a threshold value. The threshold value (1~0) signifies, to some extent, how deep into the shadow directory we would like to look for the incoming reference. A high value of say 0.95 means that we are comparing the new reference with a few of the recently referenced addresses. After a number of simulation experiments it was observed that values greater than or equal to 0.6 could be chosen as appropriate threshold values.

Pseudo code:

```
input_field ← tag_field [concatenate] set_field
if (Neural_Network_Response (input_field, set_number)
    ≥ Threshold)
then   shadow ← True for a given set_number
else   shadow ← False for a given set_number
```

Step 4: The neural network is trained, preferably on a miss only, by identifying the set to which the cache line is mapped. For the correct set n, the corresponding desired output neuron value is +1.0, whereas, all other output neurons in the neural network are assigned a value of -1.0.

Pseudo code:

```
if ( Miss() ==  True)
then { for i ← 0 to (size_of_output_layer-1)
  { if (i == set no. to which new memory reference maps)
            then desired_neuron_output[i] ← +1.0
            else desired_neuron_output[i] ← -1.0
   }
```

```
actual_neuron_output ← Neural_Network_Response()
error_vector ← {actual_neuron_output -
                                    desired_neuron_output}
Update_Weights ( error_vector, V, W)
misses ← misses + 1
       }
```

Step 5: For GA algorithm, we need two extra bits (γ and β bits) attached to the real cache directory, as compared to the LRU. Any line that produces a cache miss is brought into the cache with its γ and β bits set to 1, if and only if the NN's response to the new reference has a value greater than the threshold value.

```
Pseudo code:
if ( Miss() == True && set_number == correct)
then { if (shadow == True)
          then { γ ← 1 ; True
                 β ← 1 ; True
                   }
         else   { γ ← 0  ; False
                  β ← 0 ; False
```

```
                    }
            }
comment: γ and β fields store the status of a line.
```

Step 6: To replace a line on a miss for the case when all entries within a priority width have γ=1, we simply discard the LRU entry/line. All the other stack entries are moved down one position. The new line then takes the most recently referenced line position.

Pseudo code:

```
if ( Miss() == True && set_number == correct)
then {if (All lines within priority width of stack
          have γ == True)
           { for j ← 0 to (Associativity-2)
                    line [j]  ← line [j+1]
              line [Associativity-1] ← new_line
              comment: new_line occupies MRU position
              update γ and  β fields for the new entry
            }
        }
```

Step 7: If at least one of the lines within the priority width have γ=0 then the way to replace a line from the cache is to start from the bottom of the stack (LRU entry positions). If its γ bit equals one then transfer the line to position one (most recently referenced entry position) of the stack and set its γ bit to zero (i.e. take away the preferential status of the line). Move all other entries down one position and repeat Step 7 until an entry with γ=0 is encountered. Discard the cache line with γ=0.

Pseudo code:

```
if ( Miss() == True && set_number == correct)
then { if (At least one line within priority width
           have  γ == False)
           then { j ← 0
                while ( γ of line [j] ≠ False)
                do { Temporary ← line [0]
                for i ← 0 to (Associativity -2)
                  line [i] ← line [i+1]
                line [Associativity-1] ← Temporary
                γ ← 0 for  line [Associativity-1]
                j ← j + 1
```

```
                                    }

comment: The following 4 statements will be executed
when a line with  γ=0 is found.

                    for k ← 0 to (Associativity -2)
                          line [k] ← line [k+1]
                    line [Associativity-1] ← new_line
                    set γ and β fields for new_line
                }
        }
```

Step 8: For a cache hit, the stack update policy of the algorithm is similar to the LRU. The only difference is that the γ bit for the line currently referenced is reset to one only if the β bit was already set to one. The purpose of this step is to retain the preferential status of the line for as long as it is rereferenced. Once a line has lost its preferential status (γ=0), it must be rereferenced during its extra stack cycle in order to regain this status.

Pseudo code:

```
if ( Miss() ≠  True && set_number == correct)
then { Temporary ← line [hit_position]
            for i ← (hit_position) to (Associativity-2)
                  line [i] ← line [i+1]
            line [Associativity-1] ← Temporary
            if ( β == 1 for line [Associativity-1] )
               then      γ ← 1
               else      γ ← 0
       }
```

5.2 Performance of the GA Algorithm for BI.DATA and TPC.DATA Trace Files

In this section, performance of the generalized cache replacement algorithm (GA algorithm) for the two trace files, discussed in section 4.1, is studied and analyzed. Figures 5.3 and 5.4 provide us with a relative performance of OPT, GA, and the LRU algorithms. A backpropagation neural network paradigm that was used for guiding the replacement decisions in GA algorithm was made up of hidden and output layer neurons

incorporating TanH transfer function and Norm-Cum summation rule. This was done on the basis of our results presented in section 4.2. Also, the number of hidden layer neurons in BPNN were set to an empirically obtained value of 3/4 the number of input layer neurons/PEs. The criterion for doing as such is determined by balancing the conflicting goals of achieving good neural network learning with lower number of neurons for the given address trace files. Figure 5.4c illustrates the situation where the peak performance of GA algorithm for TPC.DATA trace file can be seen (Cache sets=2048, Associativity=8, and Line Size=8). Observe that, for the given trace files, GA algorithm was able to provide a peak differential performance of 13.88% over the LRU. This compares well with the 16.47% improvement provided by NEA algorithm for the non-estimating category of NNs. The performance of GA algorithm was found to be relatively better than the LRU for almost all the simulated cache configurations. Reasonably good performance of GA algorithm relative to NEA scheme means that the newer approach would be practically more attractive to the cache designers at the present state of VLSI technology. This is because, GA algorithm can be implemented with

small sized neural networks. Coupling this with the fact that the updation of the NNs, for GA algorithm, need not be done on every memory reference, translates to a faster cache response time, lower power consumption, and smaller chip density requirements.

5.3 Statistical Evaluation of Confidence on The Simulation Results

In section 5.4 of the thesis, performance of GA algorithm will be discussed for the traces from Standard Performance Evaluation Corporation, SPEC, benchmark programs. Size of such trace files generated using SPEC programs are usually very large. This puts constraints on the memory requirements for the simulating system. Also, the simulation time grows with, the increasing size of trace files, cache simulator, and neural networks. In order to alleviate the stated problems, to some extent, set sampling was performed during our simulations with some of the SPEC trace files. Set sampling is a method for estimating the behavior of the entire cache, made up of N sets, by using as few as N_s sets. The underlying concept behind set sampling is that

the activity across N cache sets is highly correlated [253]. Therefore, statistically speaking, the performance of the entire cache could be estimated by a few sets only. This means that the measurement of hit/miss ratios for a few cache sets can provide a reasonably accurate estimation of hit/miss ratios for the entire cache. If the correlation coefficient was unity, all the sets would be statistically identical and any one of them could be selected for experimentation purposes.

The problem with selecting a single set in set sampling is that the selected set may exhibit a behavior that is consistent with an outlier. This problem can be alleviated by selecting a few cache set N_s instead of just one. However, this raises additional questions: what should be an appropriate number for the sampled sets N_s ? For the given number of N_s sampled sets, how much confidence can be placed on the results ? Some work in this area was done by Puzak [31]. With the primitive trace files, Puzak simulated an ensemble of caches and found that approximately 10% of sets were sufficient for reasonably accurate results. Validity of Puzak's assertion is evaluated in this section of the thesis

using statistical and simulation techniques. Confidence levels on the simulation results that will be presented in the next section are also estimated. The trace files used for evaluating the appropriate number of sampled sets N_s at a given confidence level, were derived from the SPEC programs (Appendix C).

Given a cache with M misses for a trace file consisting of T references. We define the cache miss ratio as follows:

$$m = \frac{\text{Number of cache misses}}{\text{Number of ref. in a trace file}} = \frac{M}{T} \qquad (5.1)$$

For a set-associative cache having associativity A and the number of sets equal to N, the miss ratio m_i for any set i is given by

$$m_i = \frac{M_i}{T_i} \quad ; \text{ where, } i = 0,1,2, \ldots, N-1 \qquad (5.2)$$

Where, T_i = total number of references at the i^{th} set

M_i = total number of misses at the i^{th} set

The miss ratio for the entire cache is not equal to the mean of the individual miss ratios.

$$m \neq \frac{m_0 + m_1 + \ldots\ldots\ldots + m_{N-1}}{N}$$

That is, $$m \neq \frac{(M_0/T_0)+(M_1/T_1)+\ldots+M_{N-1}/T_{N-1})}{N} \quad (5.3)$$

Since,

$$\frac{M_0 + M_1 + \ldots + M_{N-1}}{T} \neq \frac{(M_0/T_0) + (M_1/T_1) + \ldots + (M_{N-1}/T_{N-1})}{N}$$

We now start looking at the relationship between the miss ratio for the entire cache and the miss ratio for each cache set in order to evaluate the effect of set sampling.

$$m = \frac{M}{T}$$

$$= \frac{M_0 + M_1 + M_2 + \ldots\ldots\ldots + M_{N-1}}{T}$$

$$= \frac{M_0}{T} + \frac{M_1}{T} + \frac{M_2}{T} + \ldots\ldots\ldots + \frac{M_{N-1}}{T} \quad (5.4)$$

$$= \frac{T_0}{T_0}\frac{M_0}{T} + \frac{T_1}{T_1}\frac{M_1}{T} + \ldots\ldots\ldots + \frac{T_{N-1}}{T_{N-1}}\frac{M_{N-1}}{T}$$

$$= \frac{M_0}{T_0}\frac{T_0}{T} + \frac{M_1}{T_1}\frac{T_I}{T} + \ldots\ldots\ldots + \frac{M_{N-1}}{T_{N-1}}\frac{T_{N-1}}{T}$$

$$= m_0\frac{T_0}{T} + m_1\frac{T_1}{T} + \ldots\ldots\ldots + m_{N-1}\frac{T_{N-1}}{T} \quad (5.5)$$

Equation (5.5) above, shows that the true miss ratio for the entire cache is the weighted sum of the miss ratios for each cache set. The weights are, in fact, probabilities p_{ri} 's of references associated with each cache set. That is,

$$m = m_0\, p_{r0} + m_1\, p_{r1} + \ldots\ldots + m_{N-1}\, p_{rN-1} \qquad (5.6)$$

The preceding equation provides us with a relation between overall cache miss ratio and the individual miss ratios for each cache set. However, by using (5.6) it is not easy to perceive the effects of sampling N_s out of N sets. In order to clearly see the effects of sampling, we rewrite equation (5.4) as follows:

$$m = \frac{1}{T} \{ M_0 + M_1 + M_2 + \ldots\ldots + M_{N-1} \}$$

$$= \frac{1}{T} \frac{(1/N)}{(1/N)} \{ \sum_{i=0}^{N-1} M_i \}$$

$$m = \frac{(1/N) \sum_{i=0}^{N-1} M_i}{(1/N) \sum_{j=0}^{N-1} T_j} \qquad (5.7)$$

If we estimate cache miss ratio m by using N_s randomly sampled sets, then

$$\hat{m} = \frac{(1/N_s) \sum_{i} M_i}{(1/N_s) \sum_{j} T_j} \qquad (5.8)$$

where, $(i,j) \in$ a set of N_s random numbers in the range $(0,N-1)$

In order to arrive at a more perceptive version of equation (5.8), we assume that the sampling is done in a sequence. Thus, without loss of generality, we can write equation (5.8) in the following manner.

$$\hat{m} = \frac{(1/N_s) \sum_{i=0}^{N_s-1} M_i}{(1/N_s) \sum_{j=0}^{N_s-1} T_j} \qquad (5.9)$$

Now, from equations (5.7) and (5.9) we observe that

$$\hat{m} \text{ -------> } m \quad \text{as} \quad N_s \text{ ---------> } N \qquad (5.10)$$

That is, the estimated miss ratio converges to true miss ratio as the number of sampled sets increases from 1 to N. However, if the cache sets are statistically identical, we can rewrite equations (5.7) and (5.9) as equations (5.11) and (5.12).

$$m = \frac{(1/N)\sum_{i=0}^{N-1} M_i}{(1/N)\sum_{j=0}^{N-1} T_j} = \frac{\cancel{(1/N)}\ \{M_\gamma \times \cancel{N}\}}{\cancel{(1/N)}\ \{T_\beta \times N\}} = \frac{M_\gamma}{T_\beta} \qquad (5.11)$$

Where, M_γ = total number of misses for any set in a cache.

T_β = total number of references at any set in a cache.

$$\hat{m} = \frac{(1/N_s)\sum_{i=0}^{N_s-1} M_i}{(1/N_s)\sum_{j=0}^{N_s-1} T_j} = \frac{\cancel{(1/N_s)}\ \{M_\gamma \times \cancel{N_s}\}}{\cancel{(1/N_s)}\ \{T_\beta \times N_s\}} = \frac{M_\gamma}{T_\beta} \qquad (5.12)$$

Hence, $$\frac{m}{\hat{m}} = 1 \qquad (5.13)$$

Equation (5.13) states that the true and estimated miss ratios are equal. This means that with only one cache set, M_γ and T_β can be estimated and put in equation (5.11) to get the miss ratio m for the entire cache. Equation (5.13) is only valid for caches having statistically identical sets, whereas, equation (5.9) states that as

$N_s \longrightarrow N$, $\hat{m} \longrightarrow m$

The rate of convergence, here, is determined by the distribution of cache misses across the sets. The statistical behavior of cache misses is not well understood and, therefore, it would be difficult to find appropriate N_s from equation (5.13).

From the preceding discussion, the conclusion is that in order to obtain an appropriate value for the number of sample sets N_s, we need to model cache misses by a well-defined process. The modeling will help define bounds on the simulation results with the given N_s. The accuracy of the model could be tested and evaluated by comparing its results with the simulation results.

To obtain N_s, we assume that the cache miss process is a Bernoulli process. Therefore, each address reference on a trace file has a probability h of being hit and m=1-h of being a miss. The Bernoulli process requires that the references be independent which is not entirely true for the cache miss process [21]. This means that the confidence intervals evaluated for the Bernoulli model will be smaller than the actual confidence intervals when statistical dependencies are taken into consideration. Therefore, the values for N_s, for different cache sizes, would only

represent lower bounds on the actual values. For the Bernoulli cache miss process, we define the number of misses as a random variable M. Thus, for a trace file containing T references, the mean and variance for the random variable M would be: $E(M) = m\,T$ and $Var(M) = m\,T\,h$. For large T, normal approximation states that the probability for M in the interval (a , b) is

$$P(a \le M^* \le b) = \Phi(b) - \Phi(a)$$

where, $M^* = \dfrac{M - E(M)}{\{Var(M)\}^{1/2}}$ = Standardized M

For normal approximation, we need to standardize M. The precise reason for doing as such is provided by the Central Limit Theorem. An intuitive answer is as follows;
Suppose that for a binomial random variable M with T trials and probability of success m, we could approximate $P(M \le 10)$ by the normal curve at 10, which is $\Phi(10)$. The approximation would be the same $\Phi(10)$ no matter what values are chosen for T and m. But, we know that $P(M \le 10)$ is different for different T and m. Therefore, the approximation could not be good except for a few T and m. Also, from the table for normal distribution, we can see that $\Phi(10) = 1.0000$, but $P(M \le 10)$ is usually not near 1.0000, and infact gets small as T increases. Clearly, we need to

bring T and m into picture somehow and, the standardization (which arises from the Central Limit Theorem) does this in an appropriate manner. Hence,

$$\Phi(b) - \Phi(a) = P\left(a \le \frac{M - E(M)}{\{Var(M)\}^{1/2}} \le b\right)$$

$$= P\left(a\{Var(M)\}^{1/2} \le M - E(M) \le b\{Var(M)\}^{1/2}\right)$$

$$= P\left(-a\{Var(M)\}^{1/2} \ge E(M) - M \ge -b\{Var(M)\}^{1/2}\right)$$

$$= P\left(M - a\{Var(M)\}^{1/2} \ge E(M) \ge M - b\{Var(M)\}^{1/2}\right)$$

$$= P\left(M - b\{Var(M)\}^{1/2} \le E(M) \le M - a\{Var(M)\}^{1/2}\right)$$

$$= P\left(M - b(m\,T\,h)^{1/2} \le m\,T \le M - a(m\,T\,h)^{1/2}\right)$$

$$= P\left(\frac{M}{T} - b\left|\frac{m\,h}{T}\right|^{1/2} \le m \le \frac{M}{T} - a\left|\frac{m\,h}{T}\right|^{1/2}\right) \qquad (5.14)$$

From equation (5.14), we are $100(\Phi(b) - \Phi(a))\%$ confident that the unknown m is infact between the limits given in equation (5.14). Equation (5.14) uses the normal distribution to fix a and b. For example, if we want 95% confidence we can fix b so that $\Phi(b)=0.975$ (That is $b=1.96$) and $\Phi(a)=0.025$ (That is $a=-1.96$). Then,

$$\Phi(b) - \Phi(a) = \Phi(1.96) - \Phi(a-1.96)$$

$$= 0.975 - 0.025 = 0.95$$

We can replace a and b in equation (5.14) by a positive valued multiplier c to get the following Confidence

Interval for the miss ratio m:

$$\frac{M}{T} - c\left|\frac{m\,h}{T}\right|^{1/2} \le m \le \frac{M}{T} + c\left|\frac{m\,h}{T}\right|^{1/2} \qquad (5.15)$$

Equation (5.15) states that the true miss ratio value for the Bernoulli cache miss process is within

$\pm\, c\left|\frac{m\,h}{T}\right|^{1/2}$ **of the estimated value** $\frac{M}{T}$.

In the case of set sampling, we choose only N_s sets out of N possible cache sets. The activity across N sets has high correlation. Therefore, for a given correlation coefficient ρ between any pair of set, we can modify the deviation term to include the effects of sampling.

$$deviation = \pm c\sqrt{\frac{mh}{T}\left(\frac{N+(N_s-N)\,\rho}{N_s}-1\right)}$$

$$deviation = \pm c\sqrt{\frac{mh}{T}\left(\frac{N}{N_s}(1-\rho)+\rho-1\right)}$$

let $N_s = \Psi N$, for Ψ = fraction of sets sampled

Then,

$$deviation = \pm c\sqrt{\frac{mh}{T}\left(\frac{1-\rho+\Psi\rho}{\psi}-1\right)} \qquad (5.16)$$

Since, the variance in equation (5.16) needs to be estimated for a large trace file for which T >> 30, therefore, the multiplier c (related to the variance) is obtained from (1-α/2)-quantile of a unit normal variate ($Z_{1-\alpha/2}$;See appendix D). Hence,

$$deviation = \pm Z_{1-\alpha/2}\sqrt{\frac{mh}{T}\left(\frac{1-\rho+\Psi\rho}{\psi}-1\right)} \qquad (5.17)$$

By making a small graph of $m(1-m)=m-m^2$ as a function of m for m between 0 and 1, we observe that the peak occurs at m=0.5. Thus, we can rewrite equation (5.17) as,

$$deviation \le \pm\frac{Z_{1-\alpha/2}}{2}\sqrt{\frac{1}{T}\left(\frac{1-\rho+\Psi\rho}{\psi}-1\right)}$$

The expression could be simplified by selecting voldman's [253] worst case correlation coefficient of 0.906 as our value for ρ. In order to match the deviation results with the simulated values for a wide variety of traces, an empirical factor of 100 is incorporated in equation (5.18) below:

$$deviation \leq \pm \frac{Z_{1-\alpha/2}}{200} \sqrt{\frac{0.094-0.094\Psi}{\Psi T}} \qquad (5.18)$$

By using equation (5.18) we computed the sampling error for compress and ear trace files as shown by figures 5.5 and 5.6, respectively. The theoretically calculated values for sampling accuracy reasonably match the simulated values for 95% confidence level as indicated by the figures. The curves also illustrate that by sampling more than 15% of cache sets ($N_s \geq 15\%$) we can be at least 95% confident that the simulation results would be accurate to within $\pm\ 2.2\times10^{-6}$ for compress trace file and within $\pm\ 7.99\times10^{-7}$ for ear trace file.

The preceding statement is true for first level unified caches ranging from 1K to 512K bytes. This is because, the caches that were simulated for comparing experimental and theoretical results ranged between the aforementioned limits. The theoretical confidence level that has been computed here, is limited by the approximations made to develop it. Due to the inherent inaccuracies in the formula developed for computing the confidence within specific theoretical bounds, approximately 30% (double) of cache sets would be sampled to get accurate results. In the next

section, set sampling technique is used for the simulations involving GA algorithm with the compress and ear trace files only. Thirty percent of cache sets would be sampled in order to obtain results for which we can state with 95% confidence that estimation errors would be bounded around low values of $\pm$ 1.45×10^{-6} and $\pm$ 5.13×10^{-7} for the respective SPEC trace files.

5.4 Study of SPEC Benchmark Performance for the GA Algorithm

In order to evaluate the performance of GA algorithm we need to test it with reliable, representative, and popular workloads. A test workload could be real or synthetic. Real workloads are sometimes referred to as benchmarks. Two benchmark trace files, TPC.DATA and BI.DATA, have already been used to evaluate the performance of NEA, EA, and conventional algorithms presented in sections 4.2, 4.3, 4.4 and, 4.5. In this section new benchmark programs/trace files from SPEC are used to evaluate the performance of GA algorithm. Standard Performance Evaluation Corporation (SPEC) is a non-profit corporation formed by leading computer vendors such as: Intel corporation, Siemens

Nixdorf, Data General, and NCR etc., to develop a standardized set of benchmarks for evaluating various performance measures related to computer systems. In January 1992, SPEC released the SPEC CINT92 and SPEC CFP92 benchmark suites. These suites consists of benchmarks derived from real end-user applications that stress and test the performance of system's processors, memory, and compilers. SPEC CINT92 contains six integer compute intensive benchmarks. Integer performance is important for systems programs, like operating system and compiler programs, as well as for application programs like word processing, databases and spreadsheets etc. SPEC CFP92 contains 14 floating point intensive benchmarks. Floating point performance is important for technical and scientific studies. A brief description of the benchmarks is given in Appendix C [254].

Address traces from the four SPEC benchmark programs: nasa7, alvinn, compress, and ear were used to drive the cache simulator incorporating GA replacement strategy. These traces, independent of the system references or interrupts, were generated on MIPS R2000 processor. The traces are currently stored in PDATS (Packed Differential Address and Time Stamp) format at the New Mexico State

University and can be downloaded from their databse at the following URL:

http://www.tracebase.nmsu.edu/

Nasa7 and alvinn were among 18 benchmark programs released by the SPEC in January 1992. Nasa7 is actually a collection of seven floating-point intensive kernels (called NAS kernels) written in FORTRAN language. They are: vpenta, cholesky, btrix, FFT, gmtry, mxm and, emit. Vpenta kernel simultaneously inverts three pentadiagonals (a routine commonly used to solve systems of partial differential equations). Cholesky performs cholesky decomposition and substitution. Btrix is a tridiagonal solver. FFT computes Fast Fourier Transforms. Gmtry is essentially a Gaussian elimination routine kernel. Mxm is a matrix-matrix multiply routine. Emit kernel creates new vortices. Alvinn is another SPEC benchmark program (written in C language) whose address traces are used in the thesis. it is a program that simulates a Backpropagation neural network.

Compress and ear are two other programs that were among the benchmark suites provided by the SPEC. Compress belongs to the SPEC CINT92 benchmark suite, whereas, ear belongs to

the SPEC CFP92 suite. Compress is a benchmark program written in C-language. This program is used in data compression applications. It reduces the size of input files by using LEMPEL-ZIV coding. Another program, ear, was also developed in C-language. It is used in medical simulations. Ear simulates the human ear by converting a sound file to a cochldagram using Fast Fourier Transforms (FFT) and other math library functions (single precision). Execution of ear program, results in intensive floating point computations and it is therefore incorporated in the SPEC CFP92 benchmark suite.

The caches considered for simulation, with nasa7 and alvinn trace files, were made up of sets (N) ranging between values of 32 and 512. For simulations with compress and ear trace files, the number of sets were varied between 32 and 2048 sets. Two different lines (L) that were simulated have sizes 8 and 16 words per line. Also, caches with three different values of associativity (A) were simulated: A=4, A=8 and A=16. Figures 5.7 thru 5.12 show the plot of miss ratio versus cache size for the nasa7 trace file (total references = 65,000,000), while figures 5.13 thru 5.18 provide similar information for the alvinn trace file (total references =59,027,112).

The graphs (5.7 thru 5.18) illustrate comparative performance of the conventional (LRU, FIFO, and MRU) and the GA algorithm. Notice that there is a general trend of GA algorithm having lower values for the miss ratios. From these graphs it is evident that the GA scheme performs well for a wide variety of cache configurations. Only for the case where cache configuration is given by L=8, A=4, and N=256 (figure 5.7), we observe that LRU (miss ratio=0.158243) barely outperforms GA (miss ratio=0.158424). This is an interesting situation since it suggests that the proposed algorithm is not invincible and that there are areas which could be explored to further improve its performance. One possible avenue that could be studied in this respect is the development of random number generators (since the initial weights of NNs are set by a random number generator). Other neural network paradigms may also be evaluated for the above mentioned purpose.

The best performance given by the GA algorithm was for the alvinn trace file (figure 5.18) when cache parameters were set at: L=16, A=16, and N=512. The performance improvement over the LRU algorithm for the best case was 8.21%. For the nasa7 trace file the best performing cache configuration (L=8, A=16, and N=512) yielded an improvement

of 6.36% (figure 5.9). Figures 5.19 thru 5.30 show the plot of miss ratio versus cache size for the compress trace file (total references ≈ 10 Million) and the ear trace file (total references ≈ 80 Million). Again, observe the general trend of GA algorithm having lower values for the miss ratios. The worst case performance of GA for the entire simulation was found to be approximately equal to the LRU. For compress, the best performance provided by GA algorithm was an improvement of 6.20% over the LRU algorithm (N=2048, A=16, L=16; see figure 5.24). This translates to a savings of 2,776 misses. For N=256, A=16, and L=16, the improvement in GA's performance was found to be 5.05% over the LRU, which translates to a savings of 11,987 misses. Even with small cache configurations and a seemingly low relative improvement of say 2.6% (figure 5.19), savings in the number of misses are considerable (56,400 fewer misses for GA over the LRU for N=32, A=4, and L=8). This is due to the fact that for small caches, high miss ratio values accompanied with relatively small improvement, translates to lower percentages of improvement. However, the savings in the number of misses are considerable even with differential percentage improvement in performance.

Due to the small size of the trace file generated from

the compress program (approximately 10 million references), large proportion of cache misses were transient misses as opposed to the steady state misses. The GA algorithm wasn't able to perform as good as it performed for the ear based trace file. This was due to the fact that the performance of GA algorithm depends on the learning of neural networks which is in turn dependent, to some extent, on the length of trace files. For ear, the best performance given by GA was an improvement of 7.3% over the conventional LRU algorithm (figure 5.27). This means that GA was able to achieve a savings of 197,255 misses over the LRU for ear based trace file consisting of approximately 80 million address references. The conventional algorithms followed a well-defined performance hierarchy rule for all the four SPEC trace files, that is, the performance of these algorithms were found to be in the following general order (from high to low): LRU, FIFO, and MRU. We believe that GA can perform even better when used for programs that references large number of memory locations.

5.5 Conclusions

This chapter began with the formulation of a theoretical frame work for a generalized neural network based cache replacement algorithm (GA algorithm). Algorithmic steps and the relevant discussion were presented next. The performance of GA was also evaluated for TPC.DATA and BI.DATA trace file. It was observed that the GA algorithm provided a consistent improvement in performance over the LRU. Best performance for GA scheme was an improvement of 13.88% over the LRU. This compares well with 16.47% improvement that NEA algorithm was able to achieve over the LRU. Trace files from four SPEC benchmark programs were also used to evaluate the performance of GA algorithm. It was observed that GA was consistently able to outperform LRU resulting in the savings of thousands of cache misses. For expediting the simulation process, we performed set sampling with two SPEC trace files, compress and ear. Reasonably accurate results were achieved by sampling 30% of cache sets. Good performance demonstrated by the GA algorithm coupled with the fact that the updation of NNs need not be done on every memory reference makes this algorithm more attractive for the practitioners involved in cache evaluation and design, as compared to the NEA/EA algorithms.

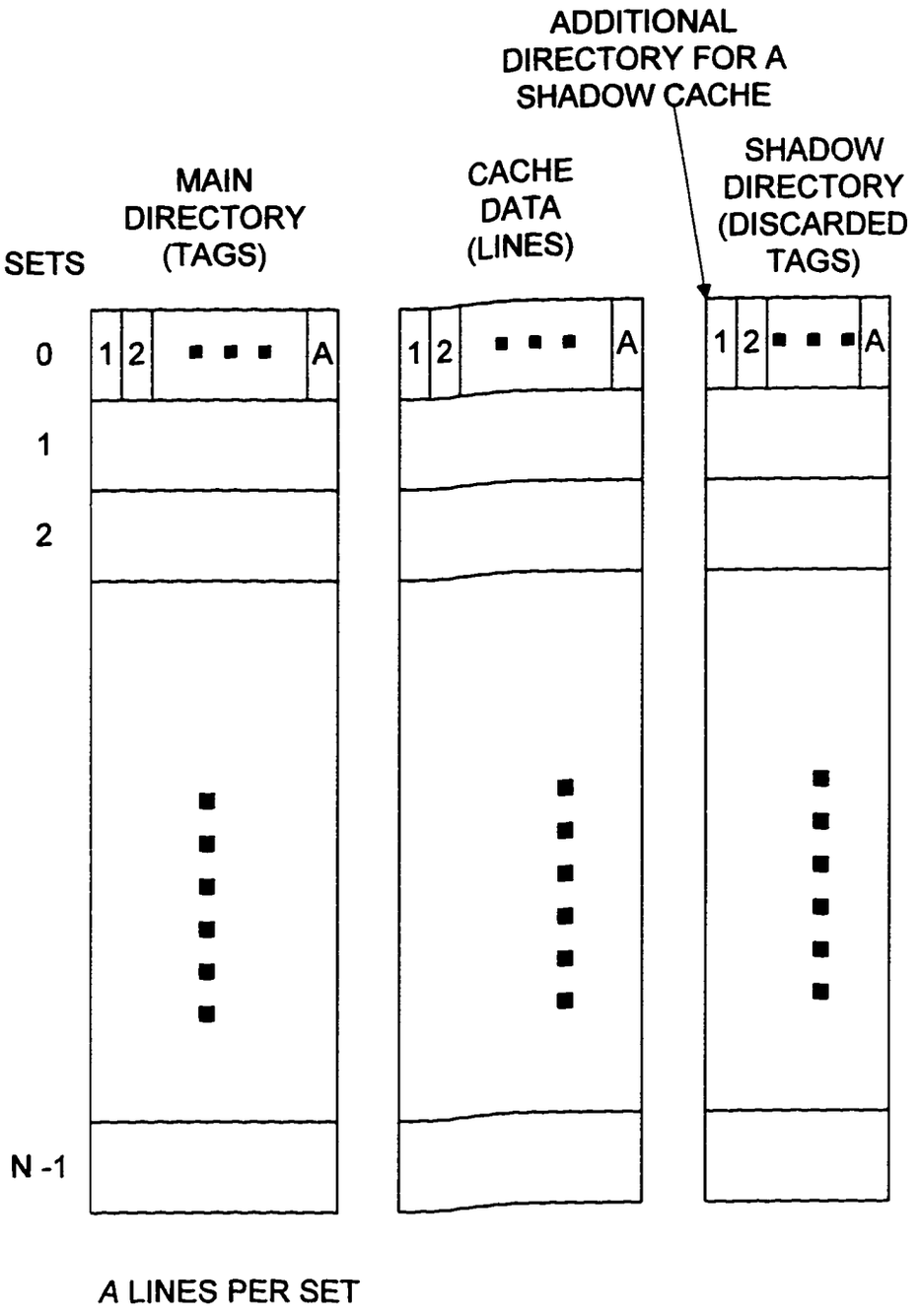


FIGURE 5.1: THE ORGANIZATION OF AN A-WAY ASSOCIATIVE CACHE WITH N-SETS AND A SHADOW DIRECTORY

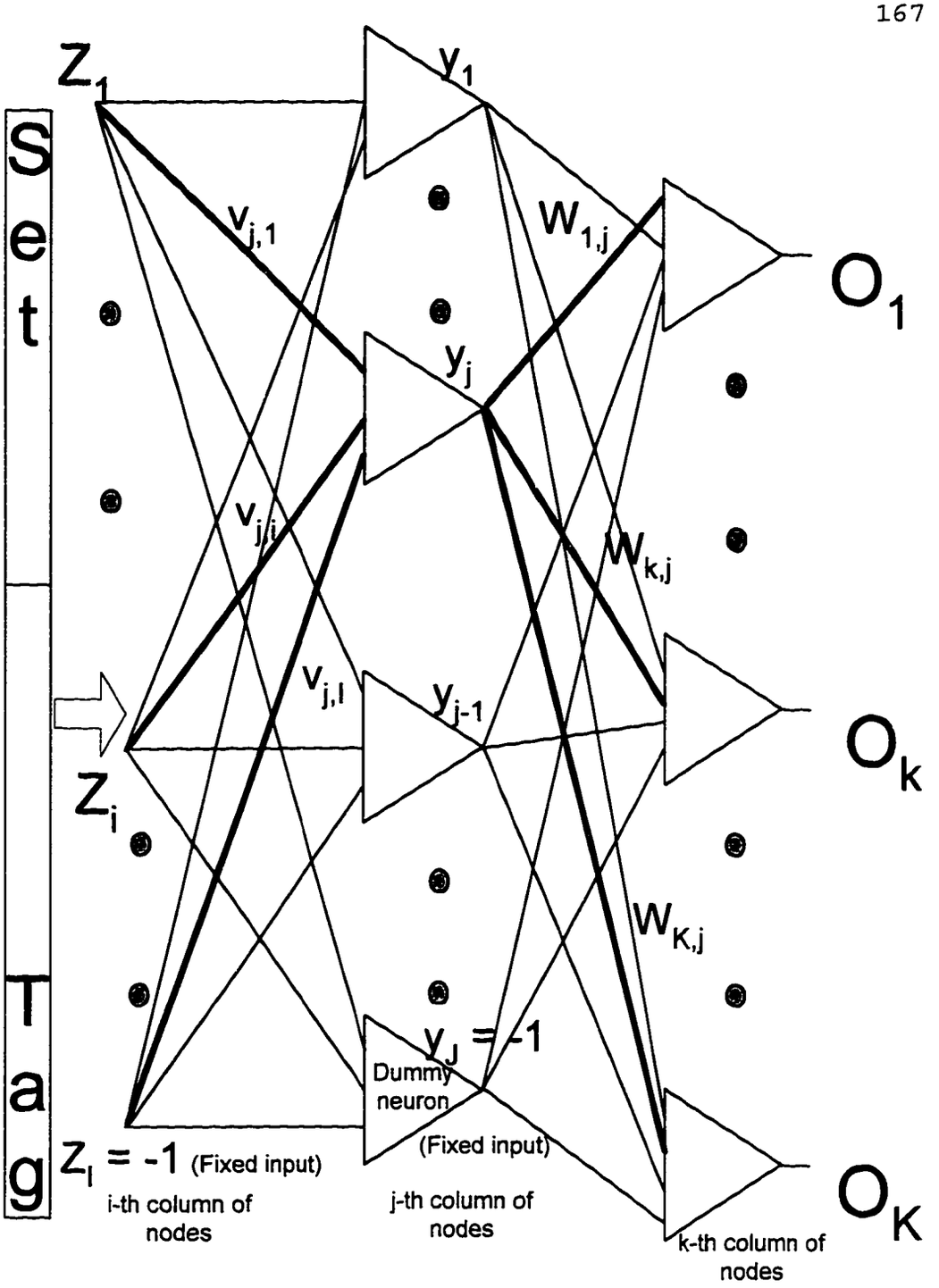


Figure 5.2: TWO LAYERED BACKPROPAGATION NEURAL NETWORK WITH SET AND TAG FIELDS OF A MEMORY ADDRESS AS AN INPUT.

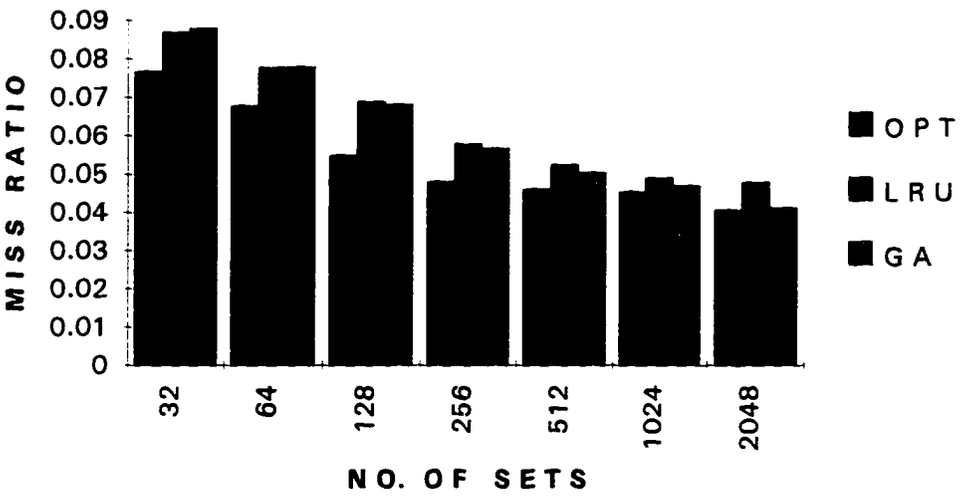


FIGURE 5.3(a): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, AND GA ALGORITHMS WITH BI.DATA AS THE TRACE FILE (ASSOCIATIVITY = 2)

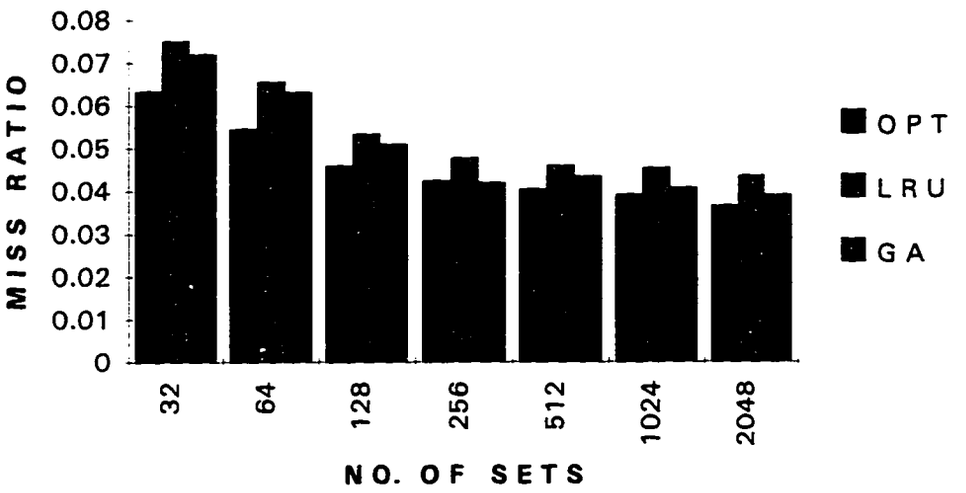


FIGURE 5.3(b): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, AND GA ALGORITHMS WITH BI.DATA AS THE TRACE FILE (ASSOCIATIVITY = 4)

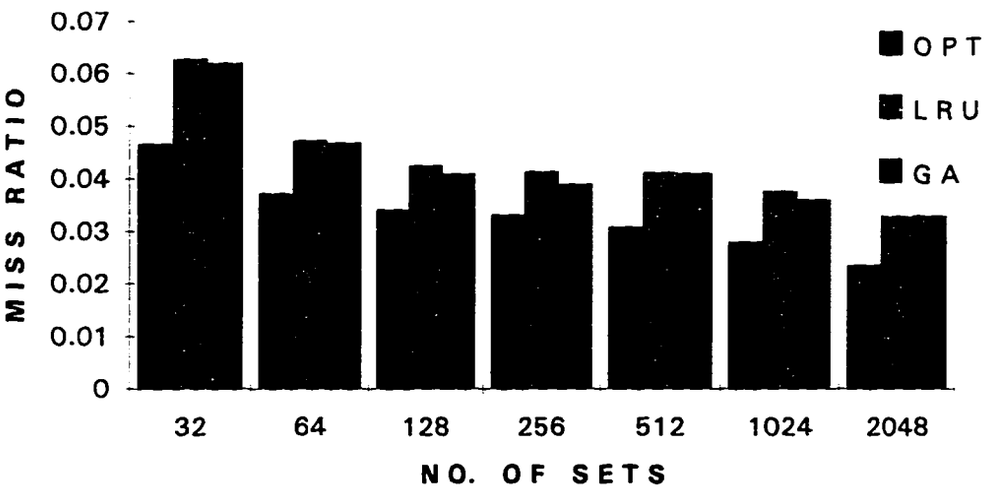


FIGURE 5.3(c): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, AND GA ALGORITHMS WITH BI.DATA AS THE TRACE FILE (ASSOCIATIVITY=8)

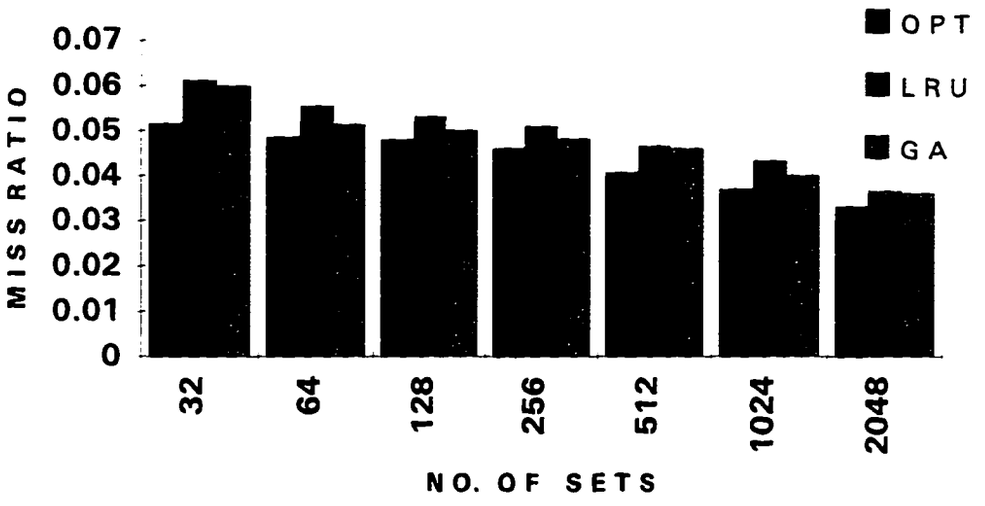


FIGURE 5.4(a): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, AND GA ALGORITHMS WITH TPC.DATA AS THE TRACE FILE (ASSOCIATIVITY = 2)

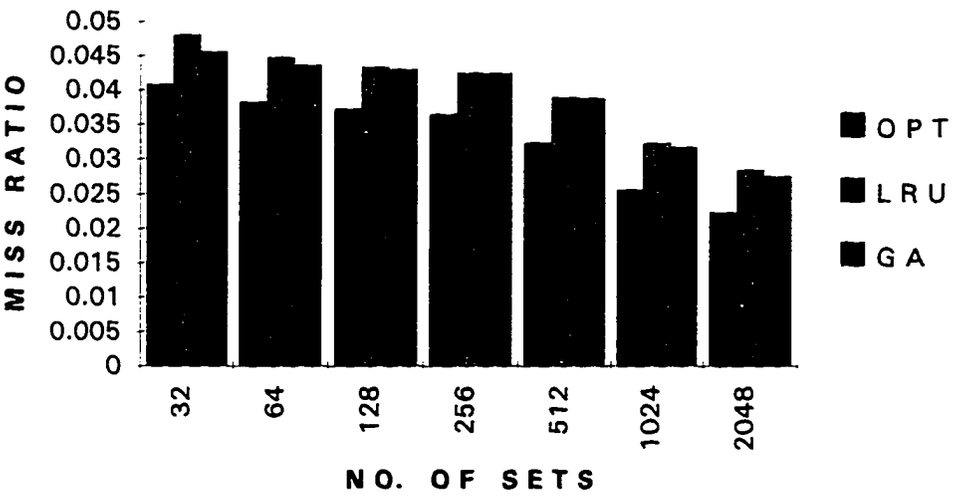


FIGURE 5.4(b): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, AND GA ALGORITHMS WITH TPC.DATA AS THE TRACE FILE (ASSOCIATIVITY = 4)

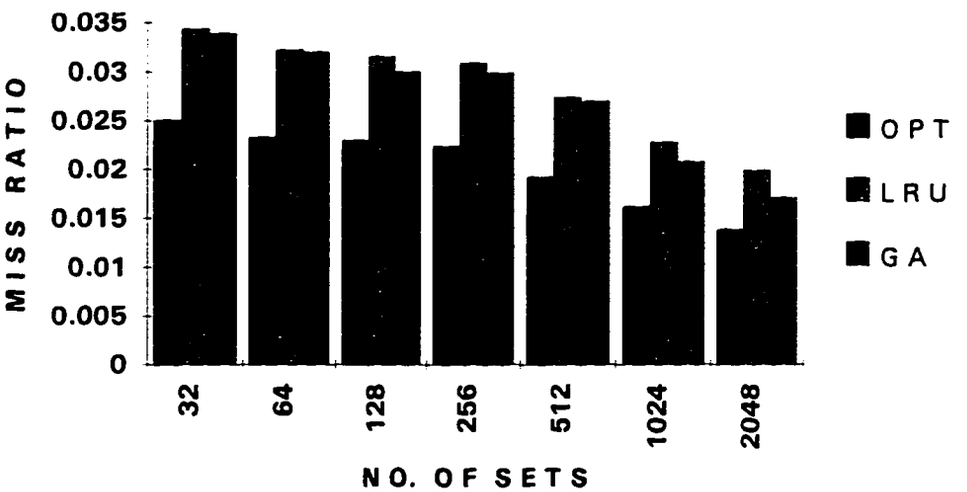


FIGURE 5.4(c): MISS RATIO vs. CACHE SIZE FOR OPT, LRU, AND GA ALGORITHMS WITH TPC.DATA AS THE TRACE FILE (ASSOCIATIVITY=8)

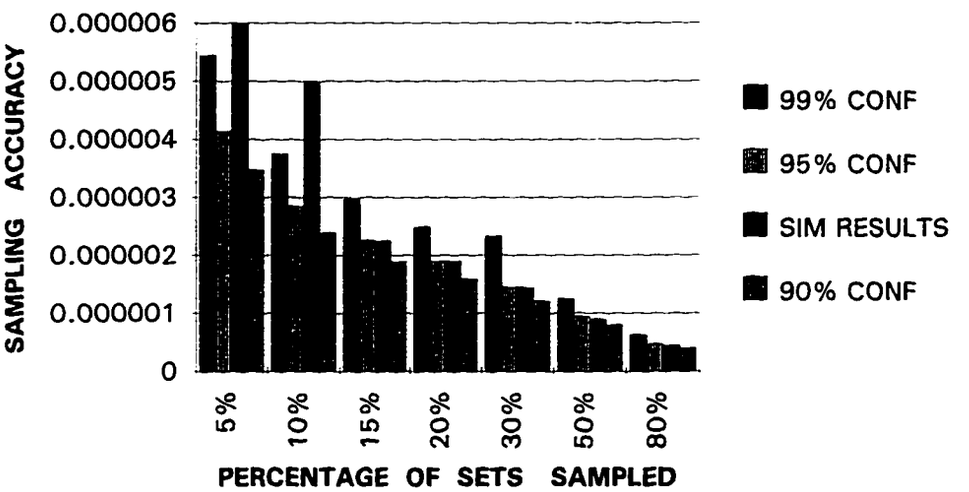


FIGURE 5.5: SAMPLING ACCURACY vs. PERCENTAGE OF SETS SAMPLED FOR COMPRESS TRACE FILE

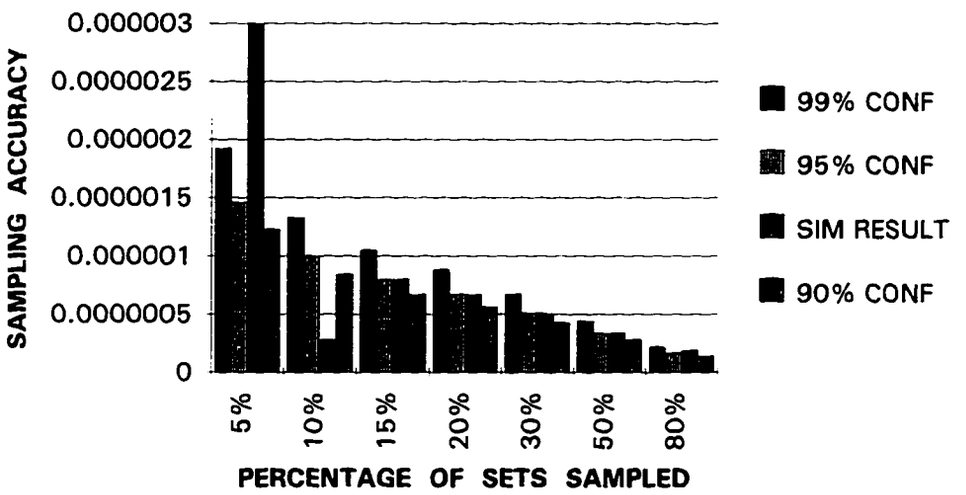


FIGURE 5.6: SAMPLING ACCURACY vs. PERCENTAGE OF SETS SAMPLED FOR EAR TRACE FILE

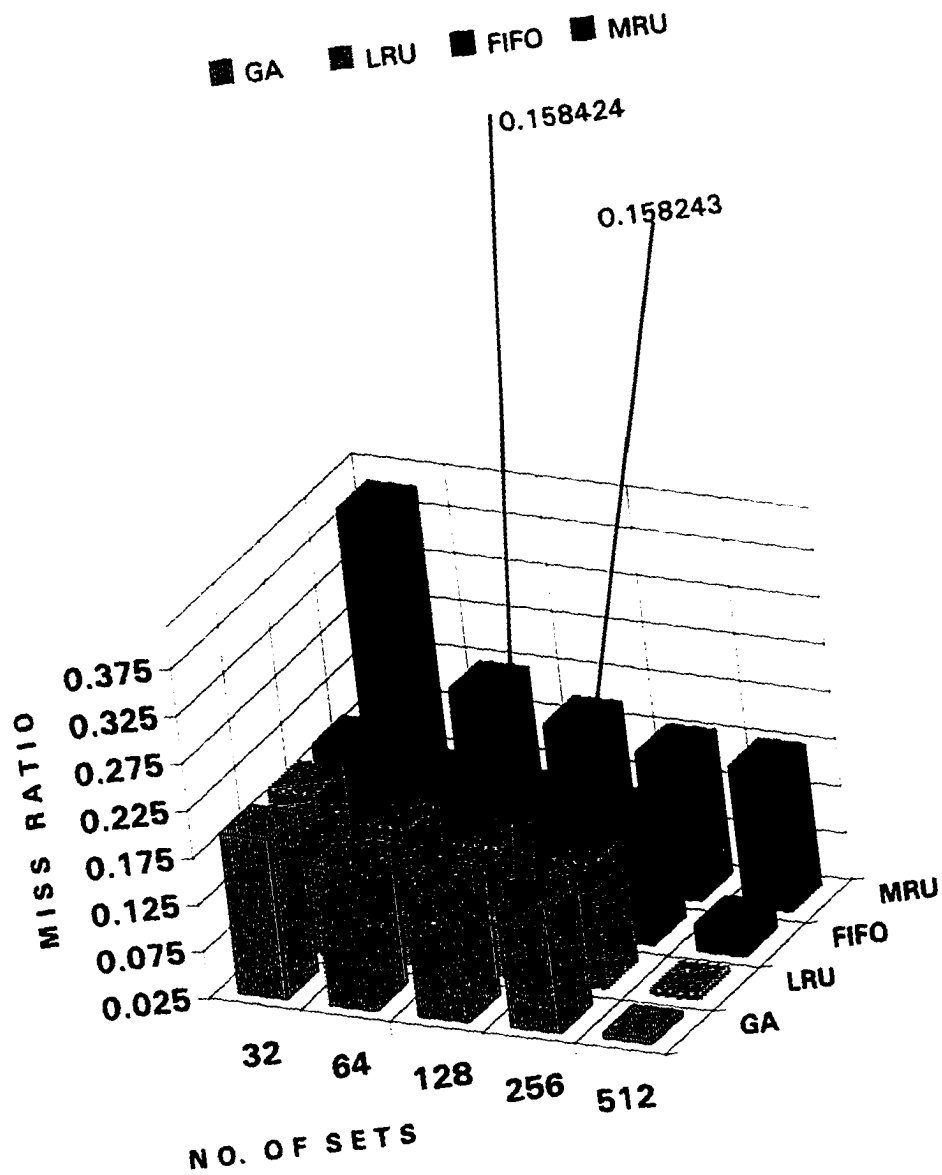


FIGURE 5.7: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH NASA7 AS THE TRACE FILE (LINE SIZE=8 AND ASSOCIATIVITY=4)

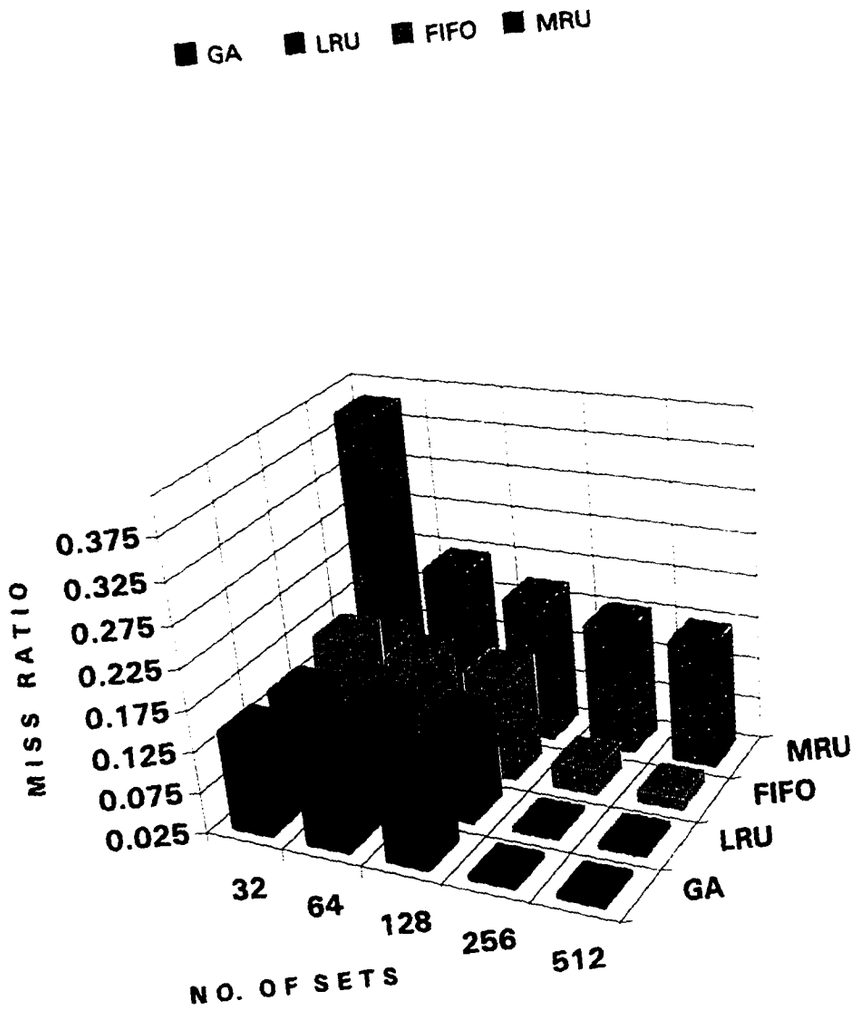


FIGURE 5.8: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH NASA7 AS THE TRACE FILE (LINE SIZE = 8 AND ASSOCIATIVITY = 8)

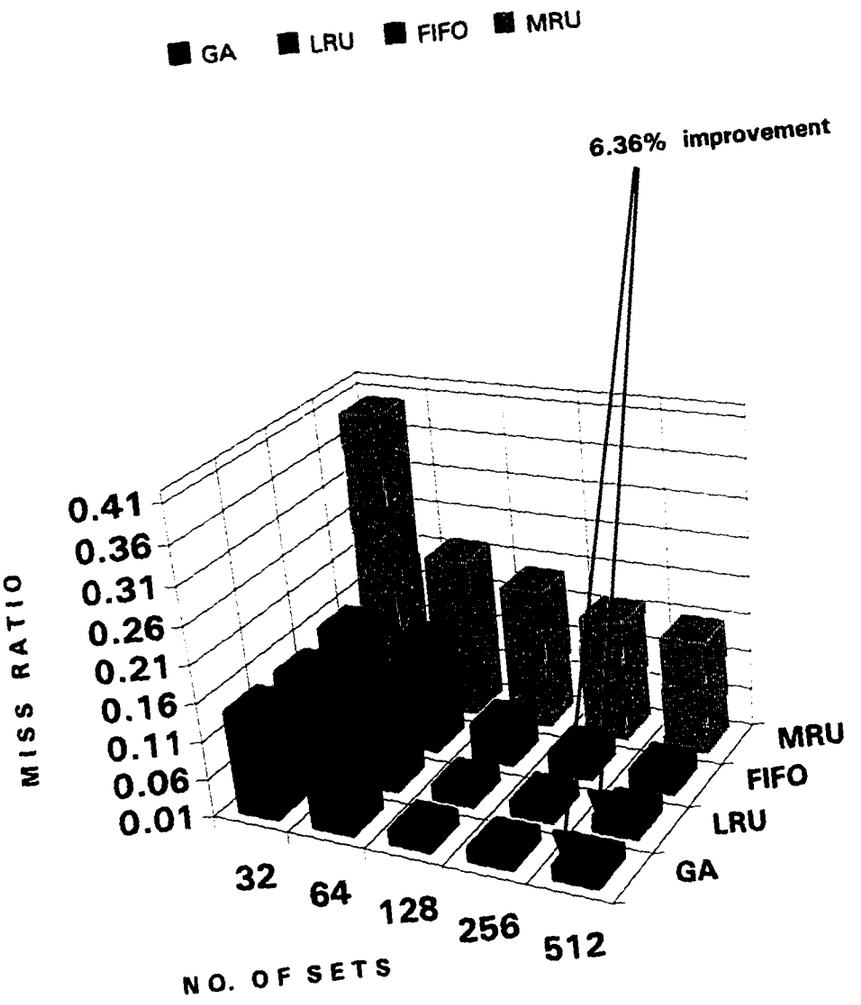


FIGURE 5.9: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH NASA7 AS THE TRACE FILE (LINE SIZE=8 AND ASSOCIATIVITY=16)

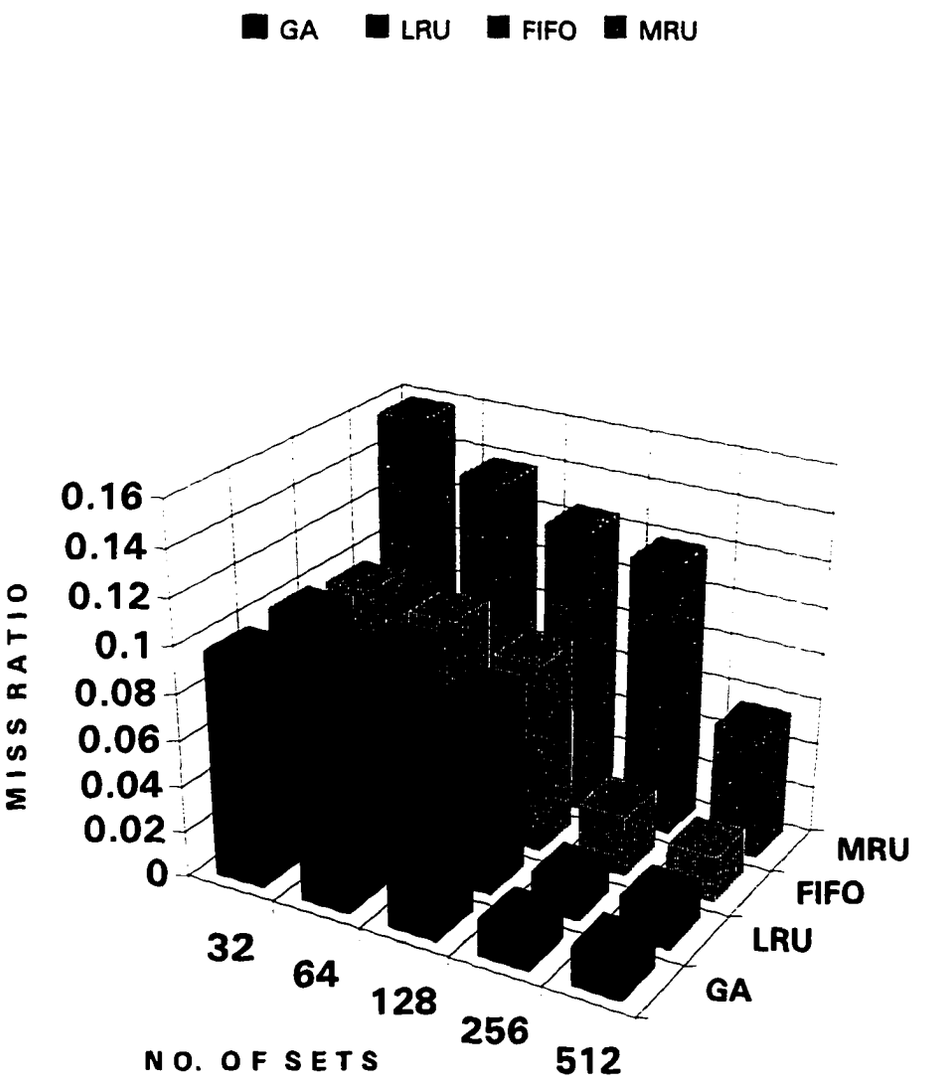


FIGURE 5.10: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH NASA7 AS THE TRACE FILE (LINE SIZE = 16 AND ASSOCIATIVITY = 4)

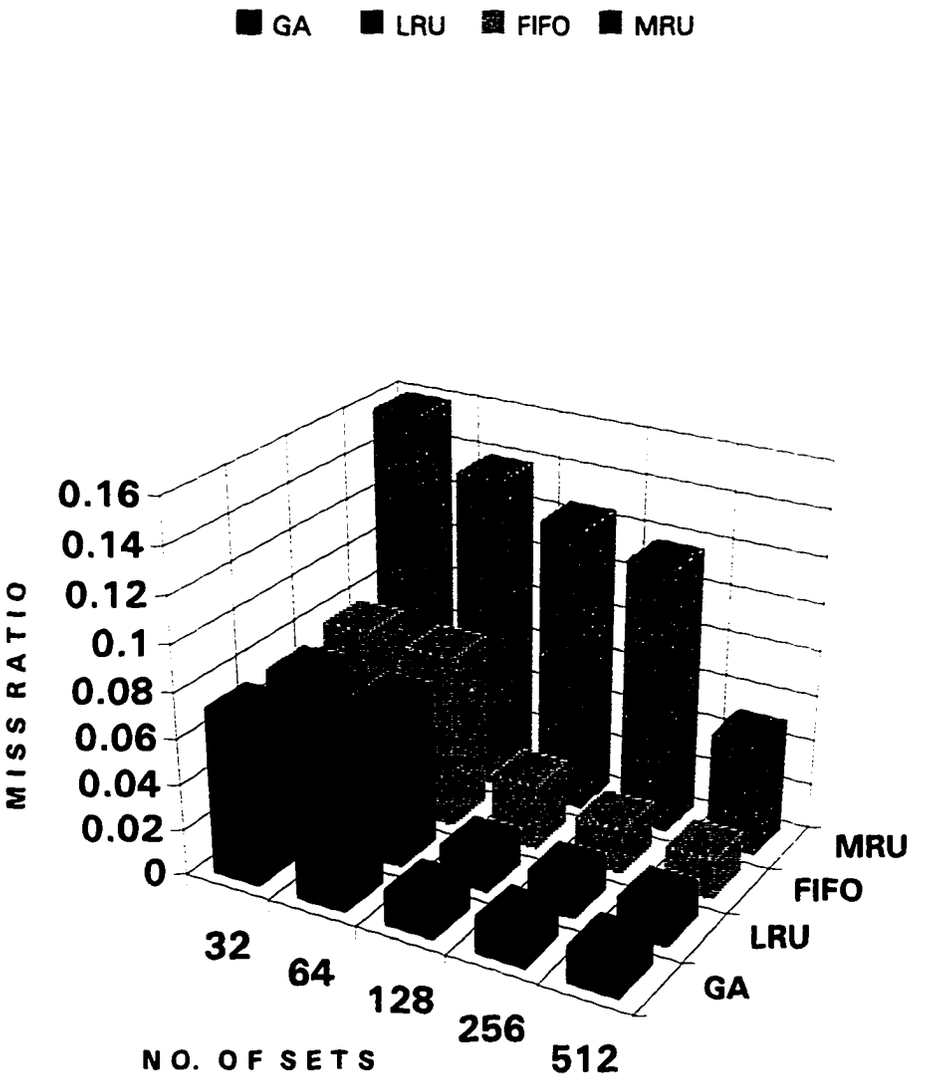


FIGURE 5.11: MISS RATIO vs.. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH NASA7 AS THE TRACE FILE (LINE SIZE = 16 AND ASSOCIATIVITY = 8)

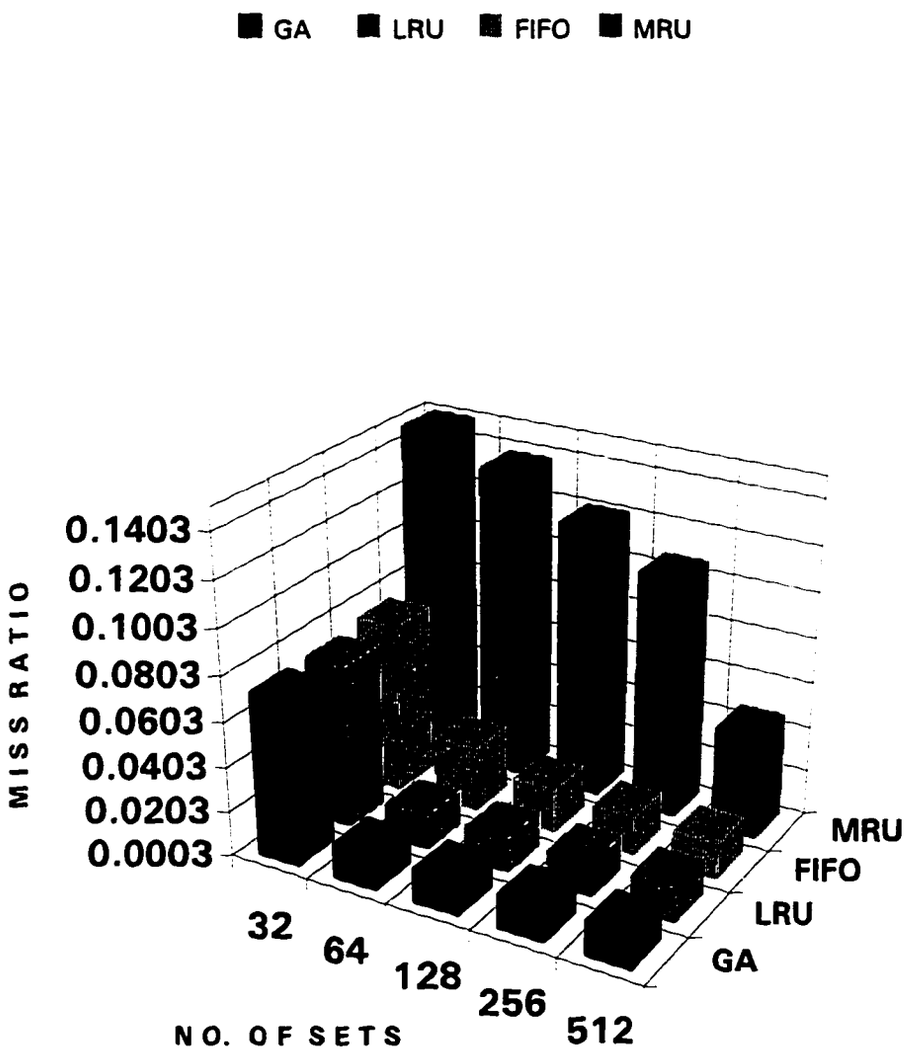


FIGURE 5.12: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH NASA7 AS THE TRACE FILE (LINE SIZE = 16 AND ASSOCIATIVITY = 16)

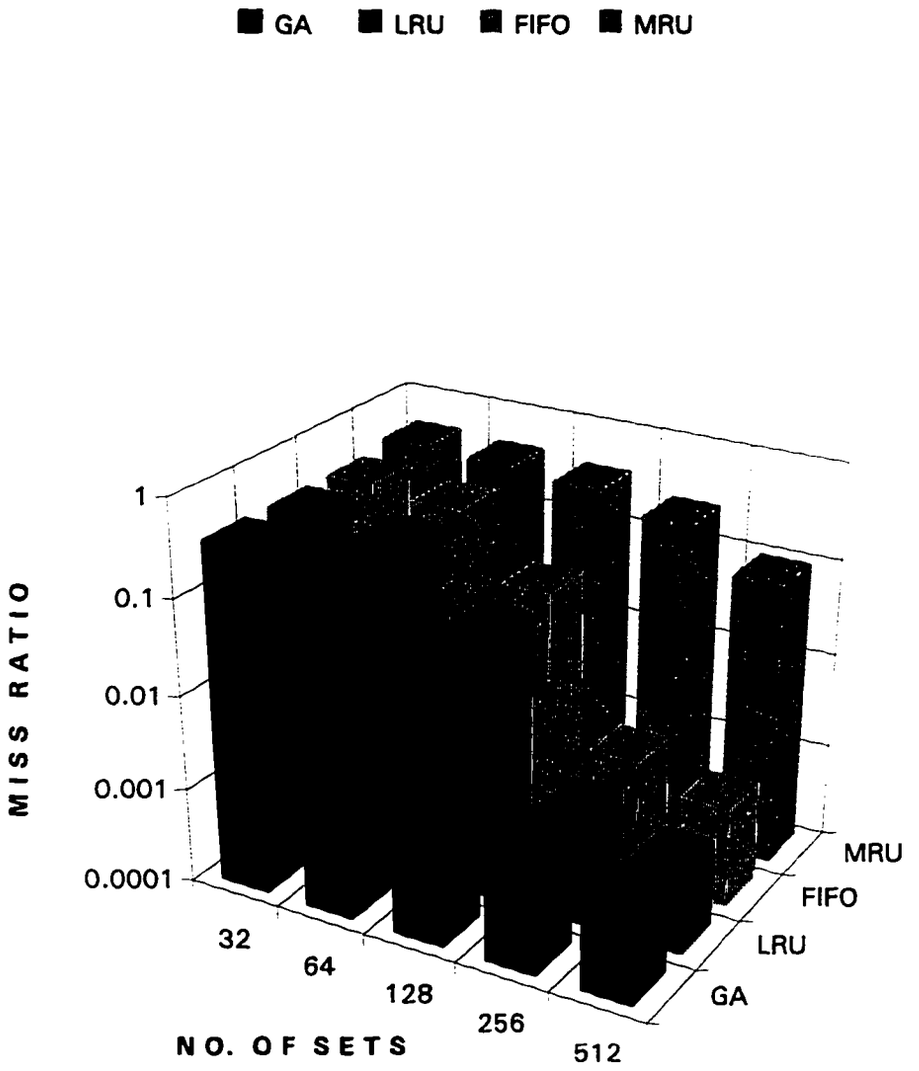


FIGURE 5.13: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH ALVINN AS THE TRACE FILE (LINE SIZE=8 AND ASSOCIATIVITY=4)

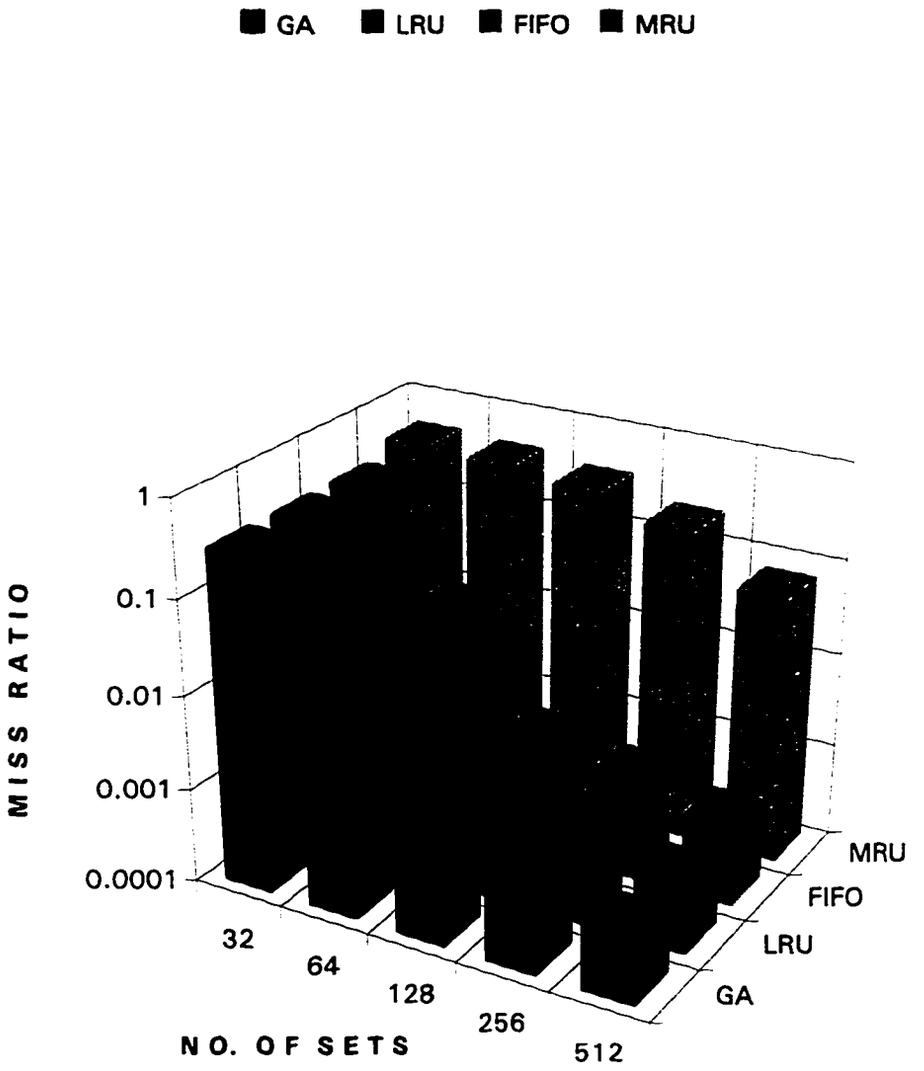


FIGURE 5.14: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH ALVINN AS THE TRACE FILE (LINE SIZE =8 AND ASSOCIATIVITY =8)

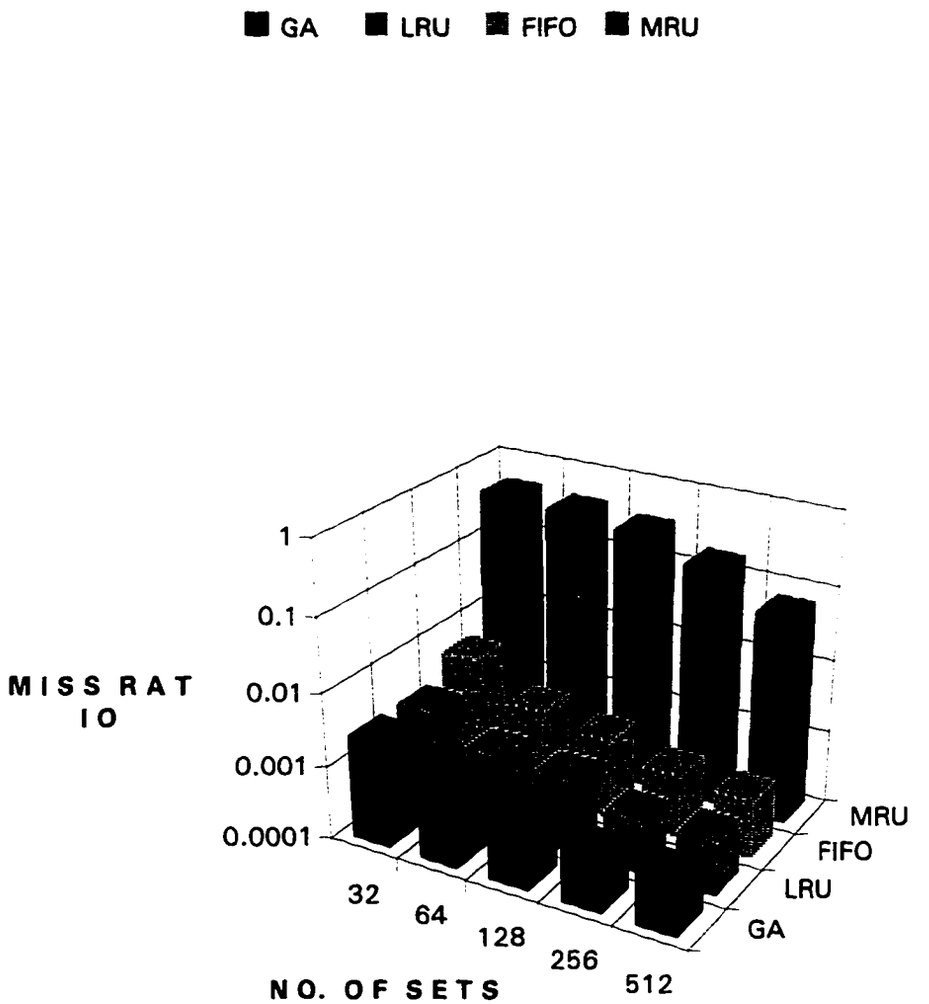


FIGURE 5.15: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH ALVINN AS THE TRACE FILE (LINE SIZE=8 AND ASSOCIATIVITY=16)

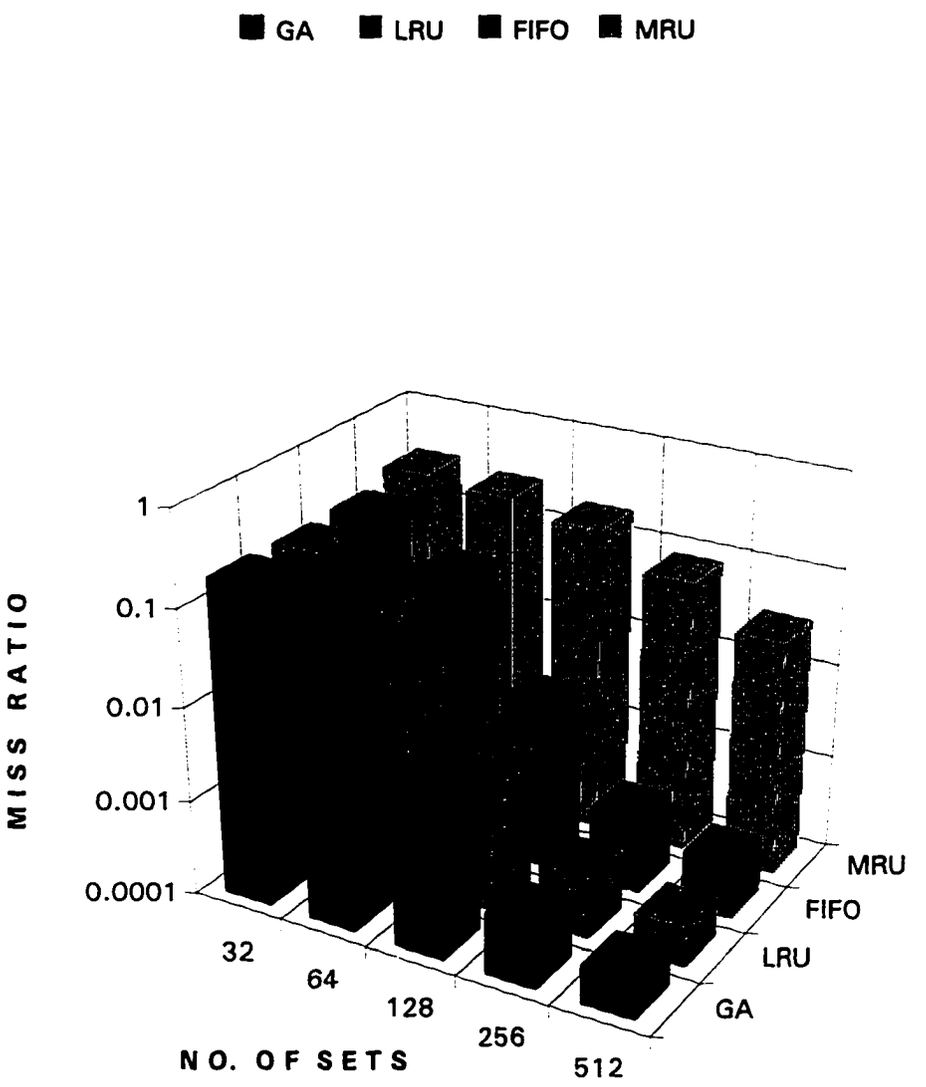


FIGURE 5.16: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH ALVINN AS THE TRACE FILE (LINE SIZE=16 AND ASSOCIATIVITY=4)

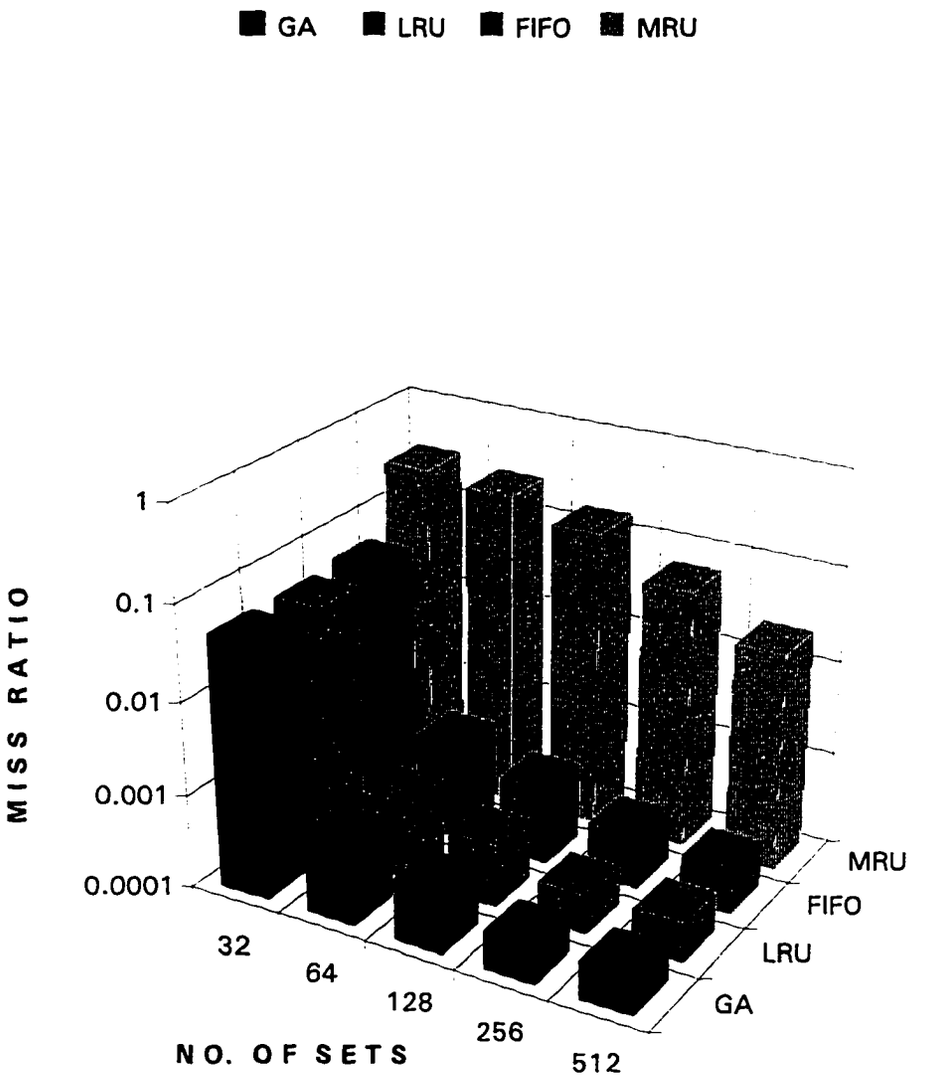


FIGURE 5.17: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH ALVINN AS THE TRACE FILE (LINE SIZE = 16 AND ASSOCIATIVITY = 8)

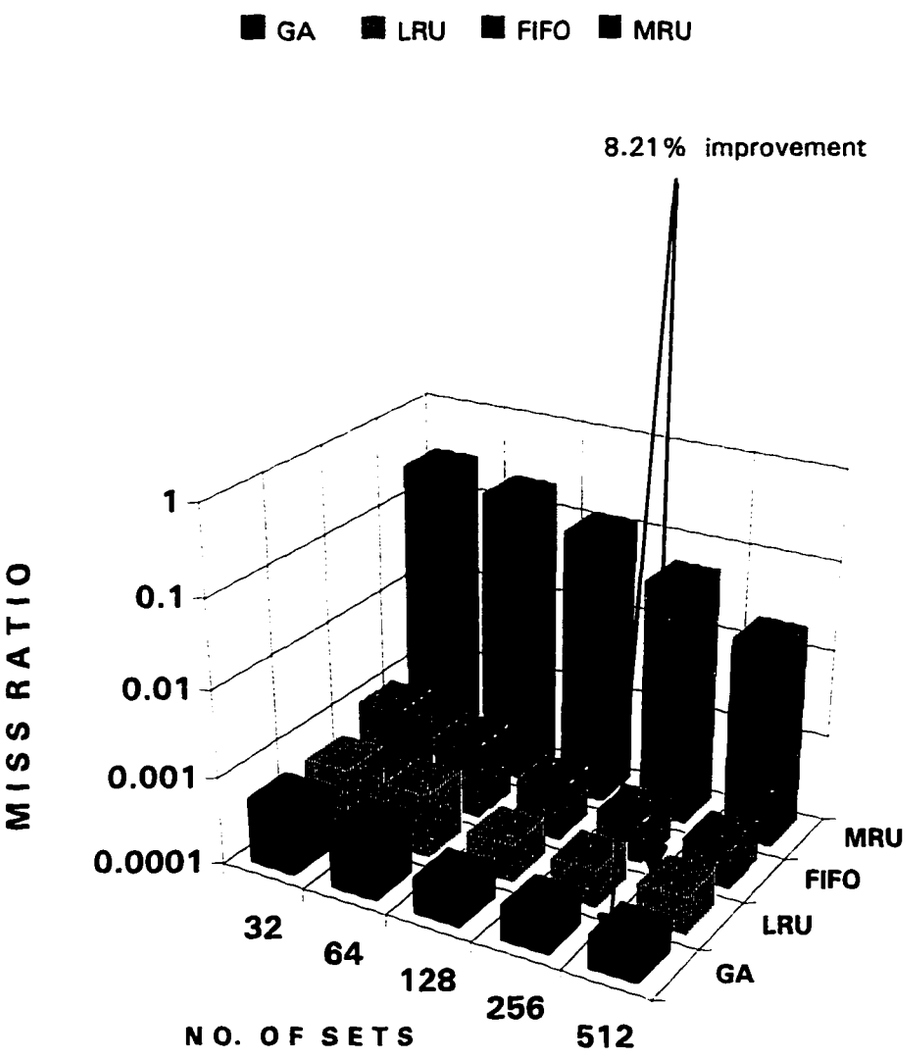


FIGURE 5.18: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH ALVINN AS THE TRACE FILE (LINE SIZE = 16 AND ASSOCIATIVITY = 16)

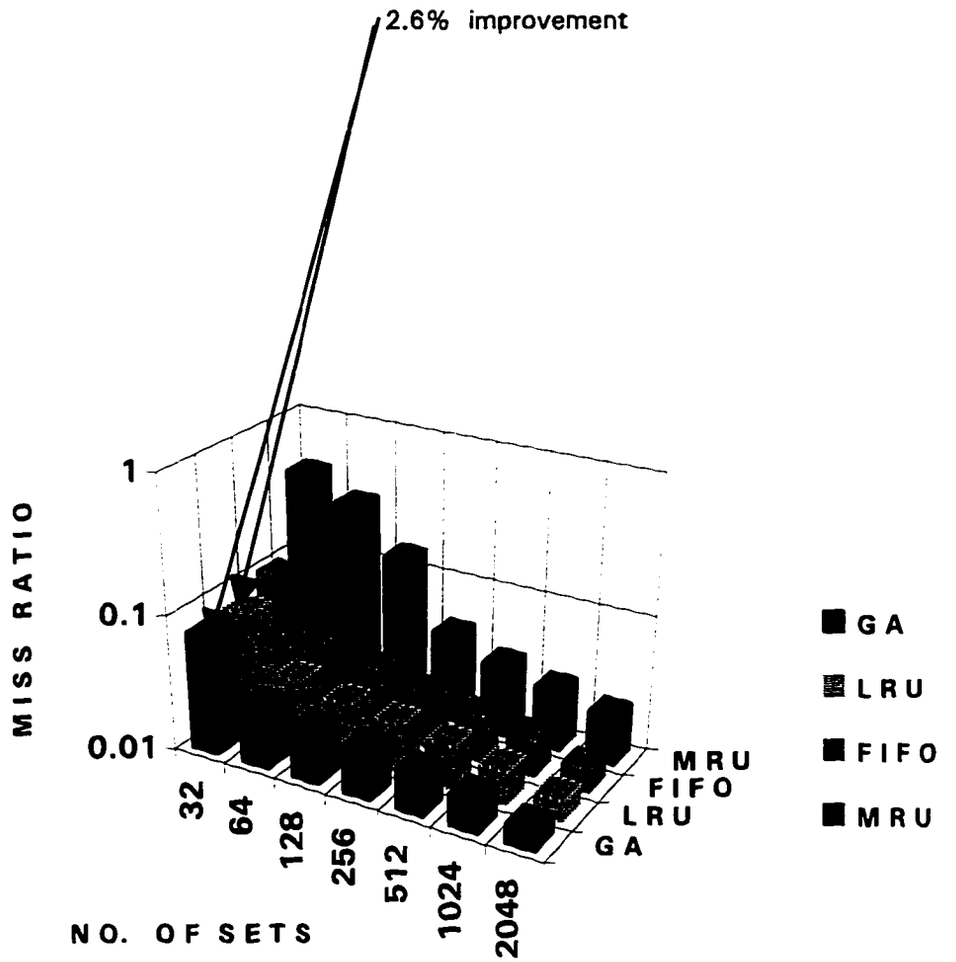


FIGURE 5.19: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH COMPRESS AS THE TRACE FILE (LINE SIZE = 8 AND ASSOCIATIVITY = 4)

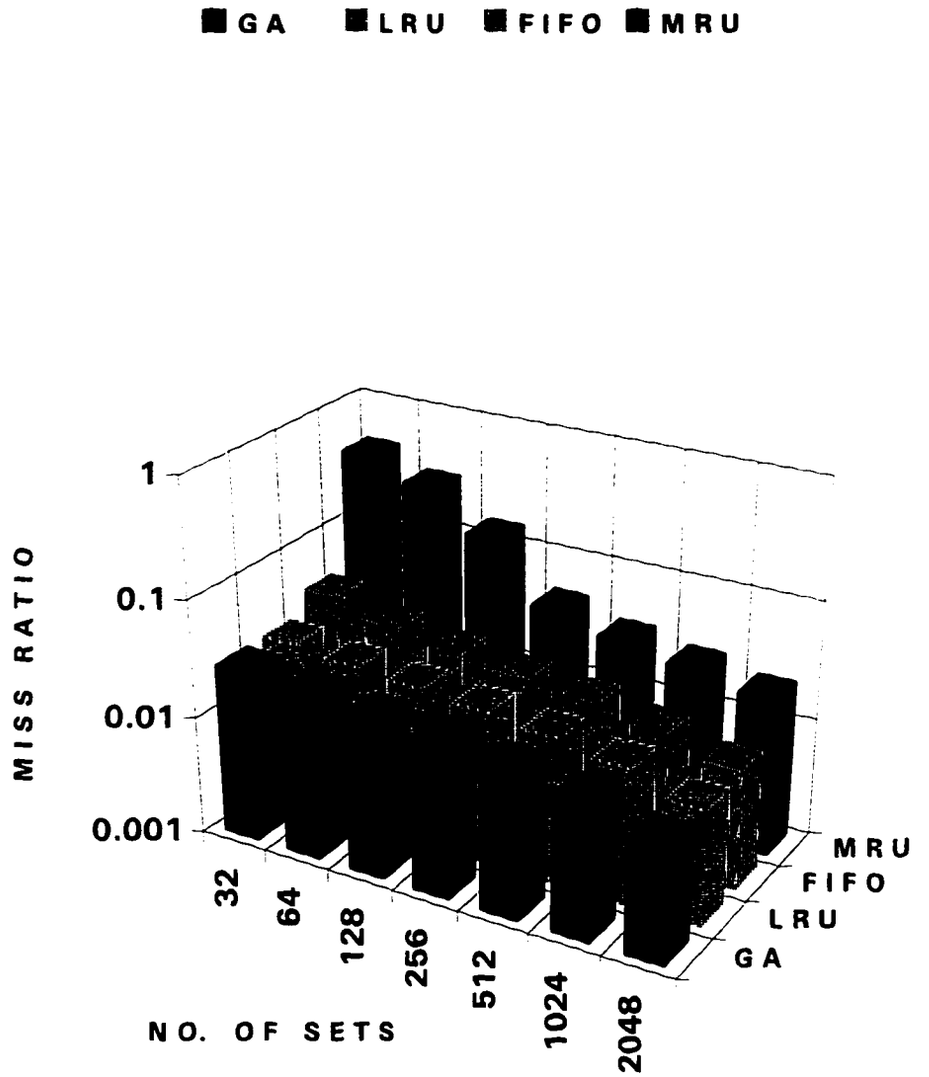


FIGURE 5.20: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH COMPRESS AS THE TRACE FILE (LINE SIZE = 8 AND ASSOCIATIVITY = 8)

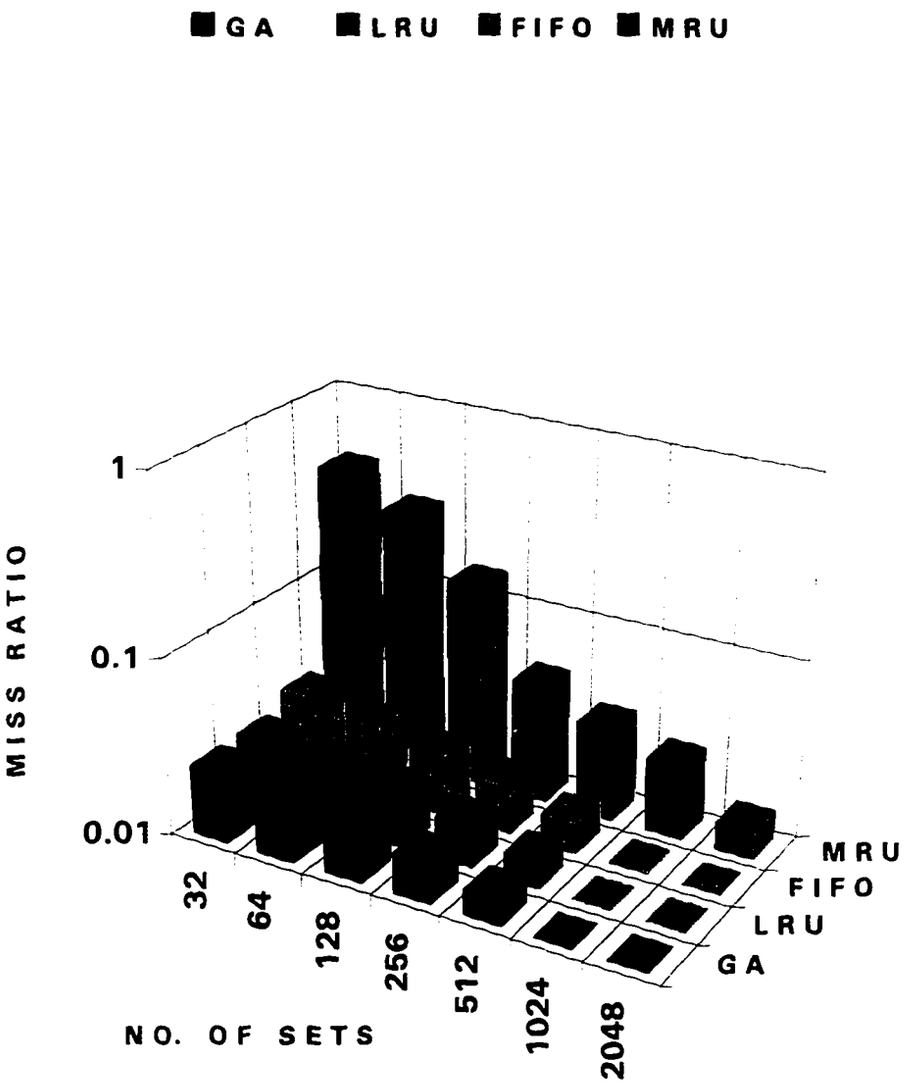


FIGURE 5.21: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH COMPRESS AS THE TRACE FILE (LINE SIZE = 8 AND ASSOCIATIVITY = 16)

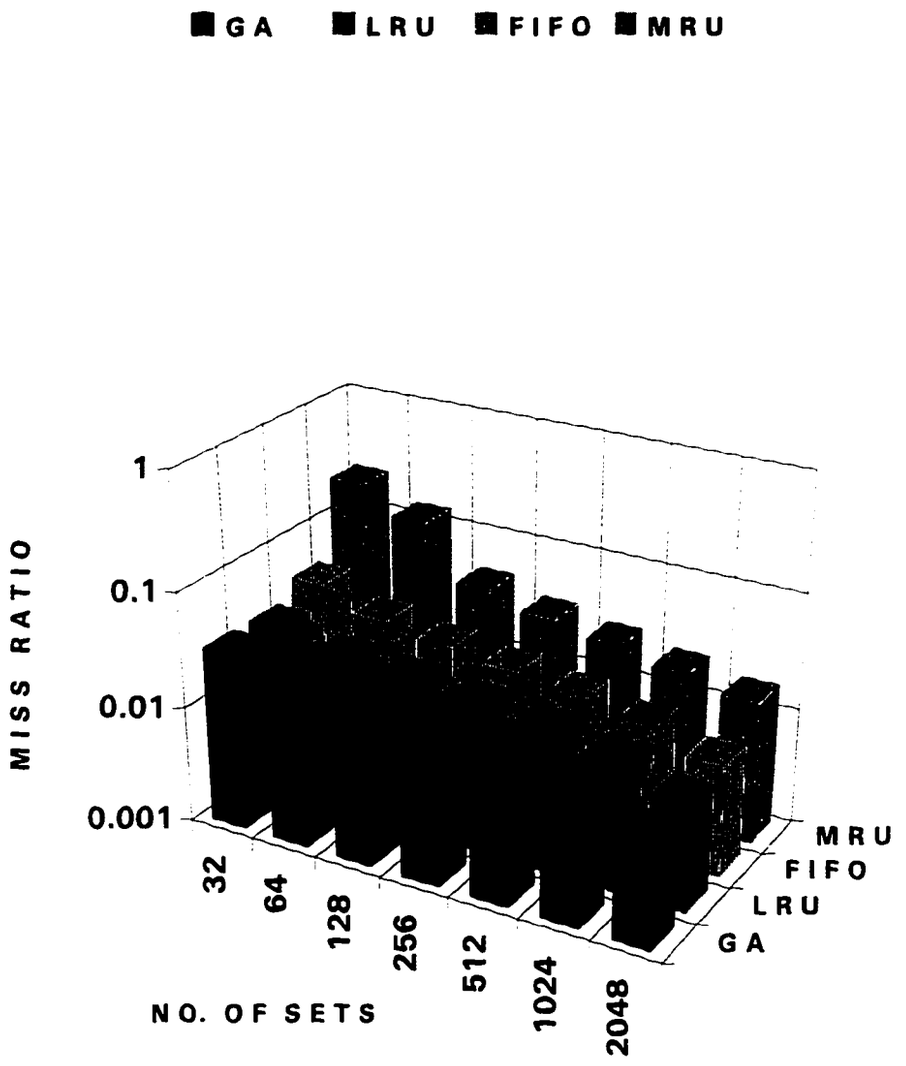


FIGURE 5.22: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH COMPRESS AS THE TRACE FILE (LINE SIZE = 16 AND ASSOCIATIVITY = 4)

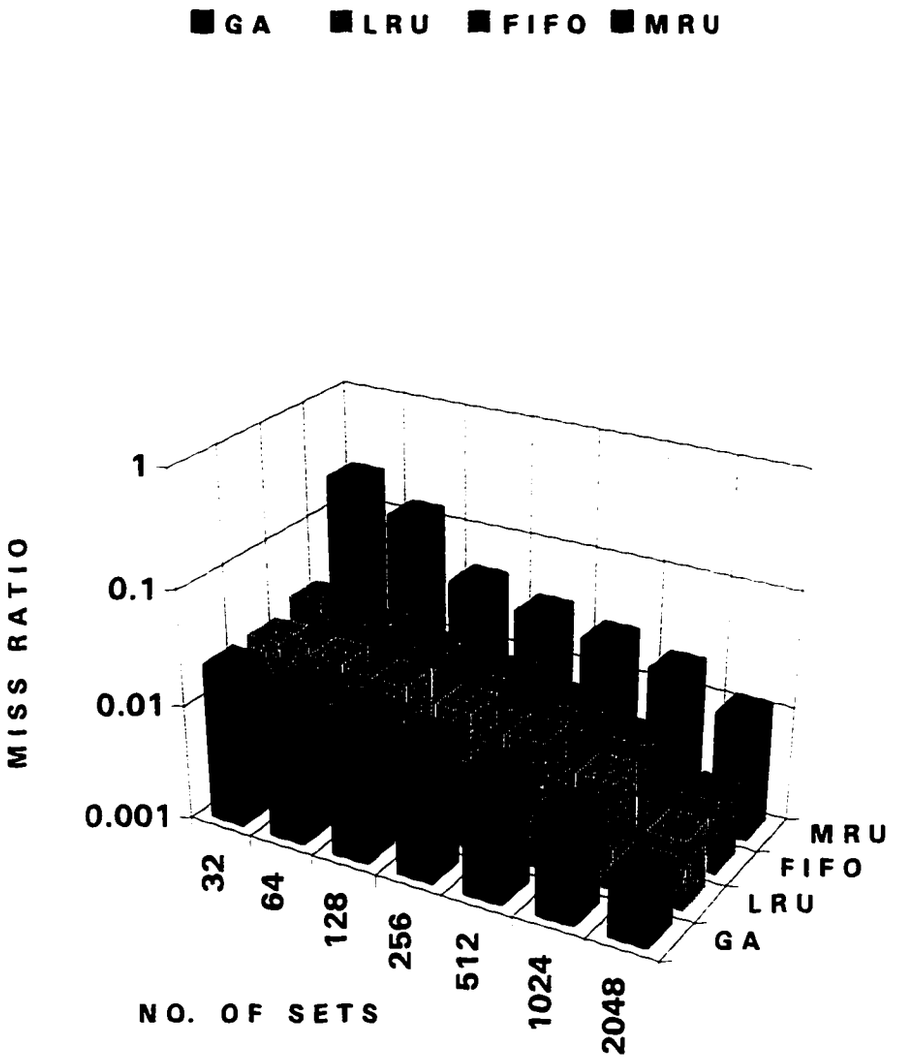


FIGURE 5.23: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH COMPRESS AS THE TRACE FILE (LINE SIZE = 16 AND ASSOCIATIVITY = 8)

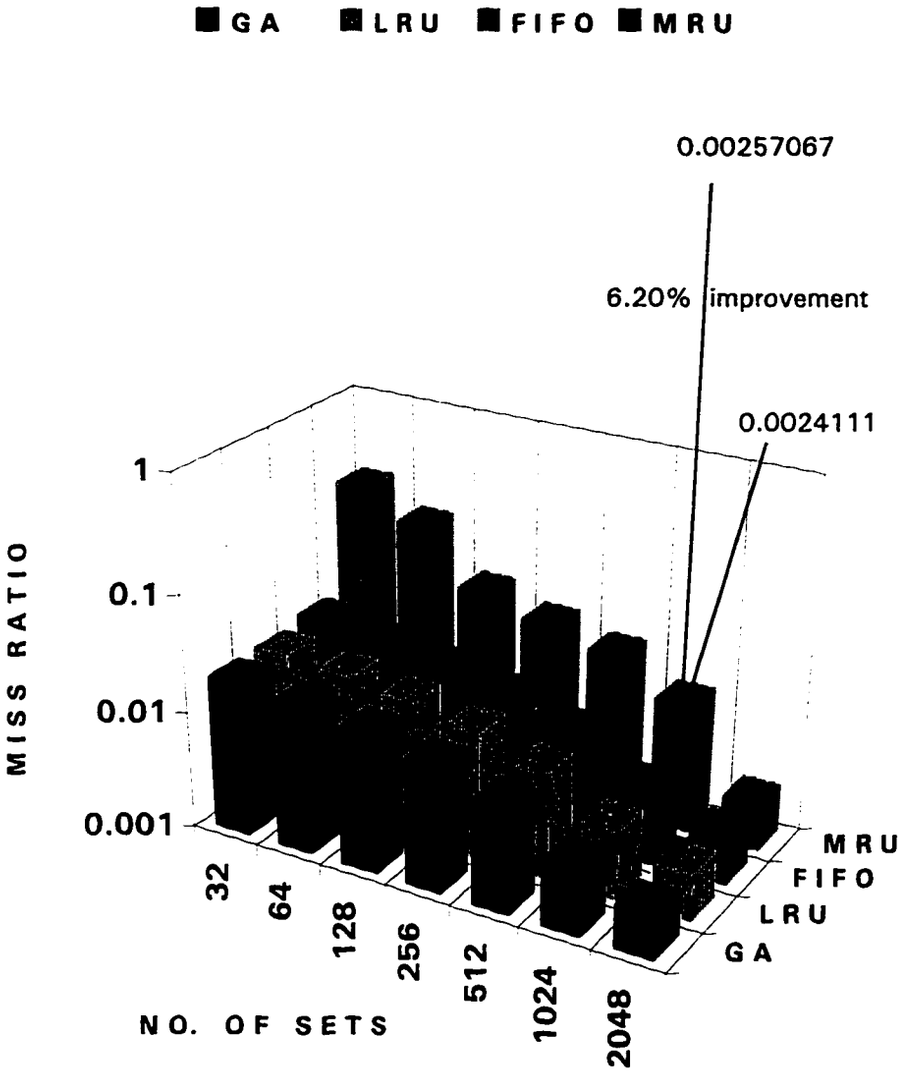


FIGURE 5.24: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH COMPRESS AS THE TRACE FILE (LINE SIZE = 16 AND ASSOCIATIVITY = 16)

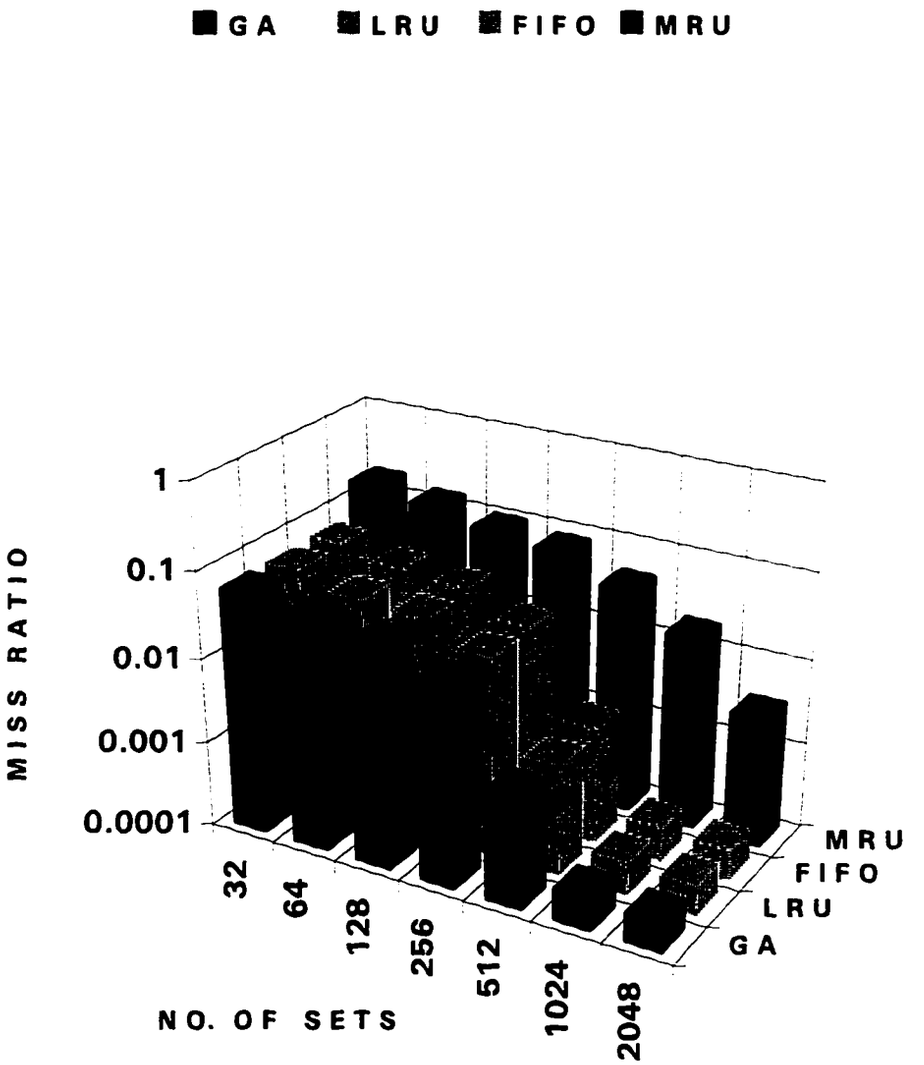


FIGURE 5.25: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH EAR AS THE TRACE FILE (LINE SIZE = 8 AND ASSOCIATIVITY = 4)

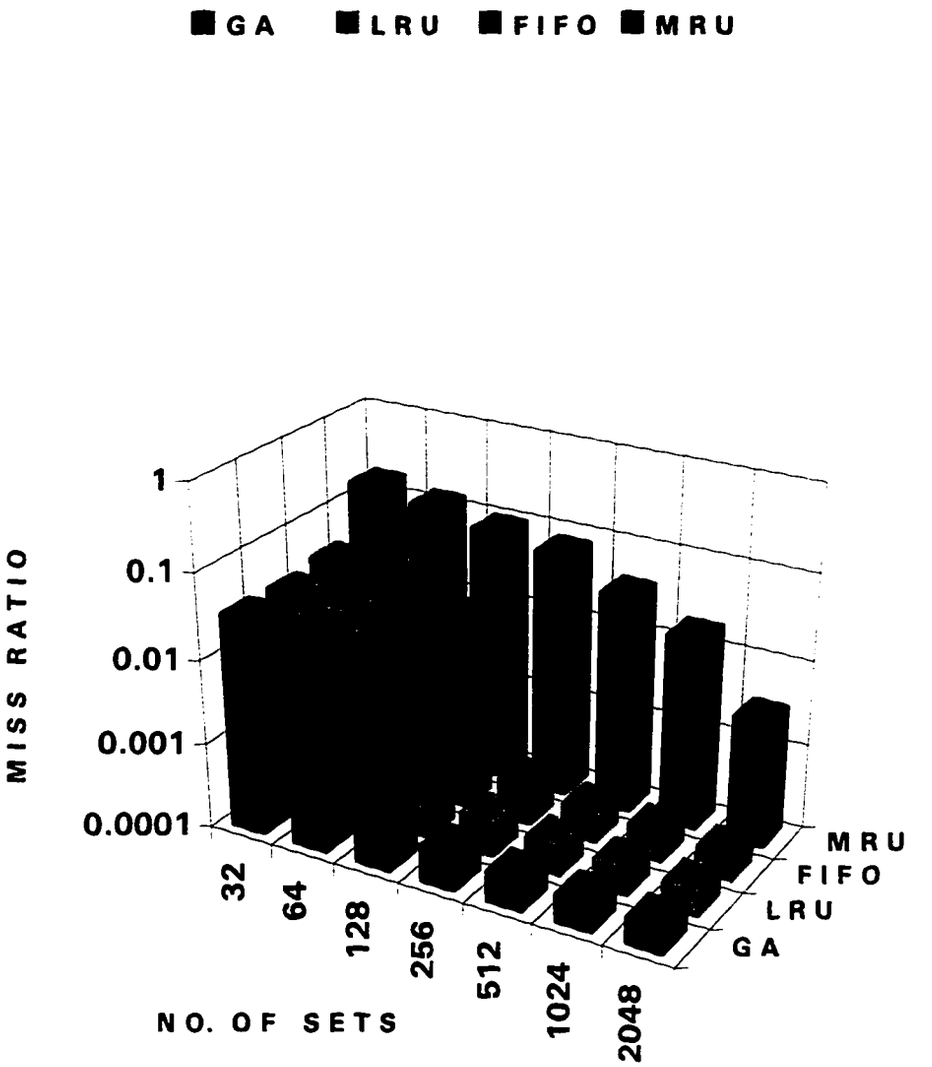


FIGURE 5.26: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND THE MRU ALGORITHMS WITH EAR AS THE TRACE FILE (LINE SIZE = 8 AND ASSOCIATIVITY = 8)

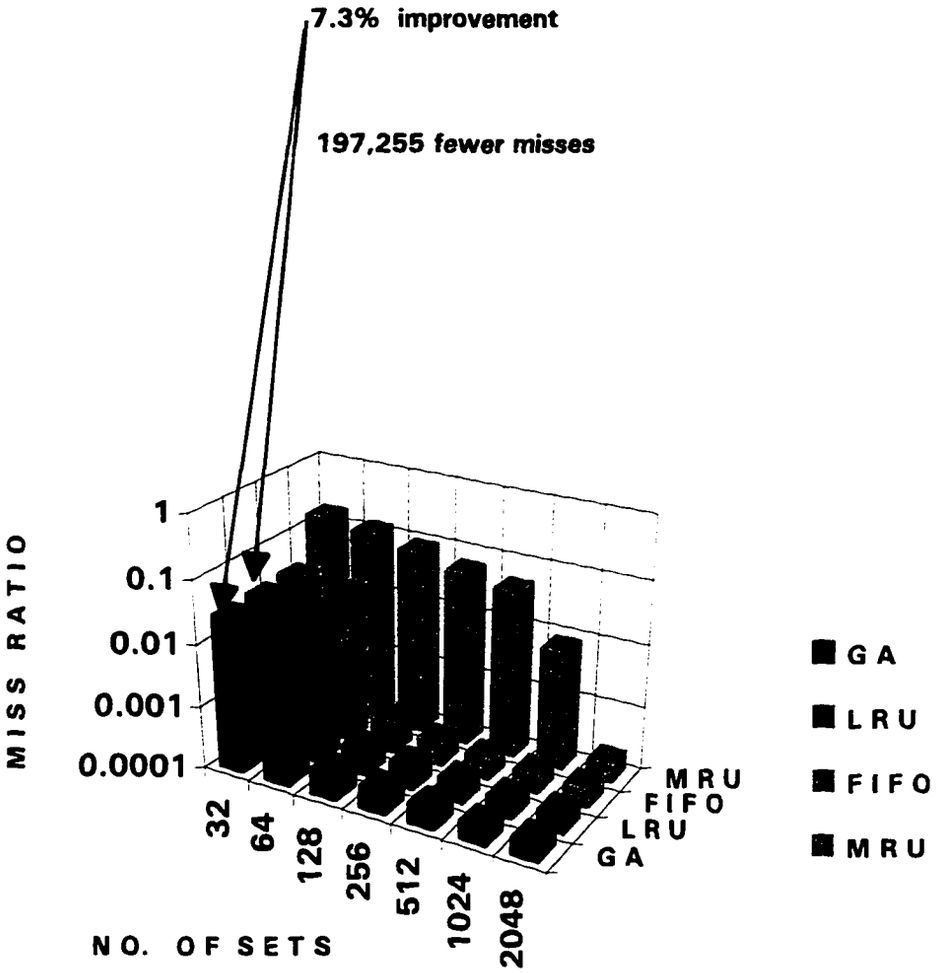


FIGURE 5.27: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH EAR AS THE TRACE FILE (LINE SIZE = 8 AND ASSOCIATIVITY = 16)

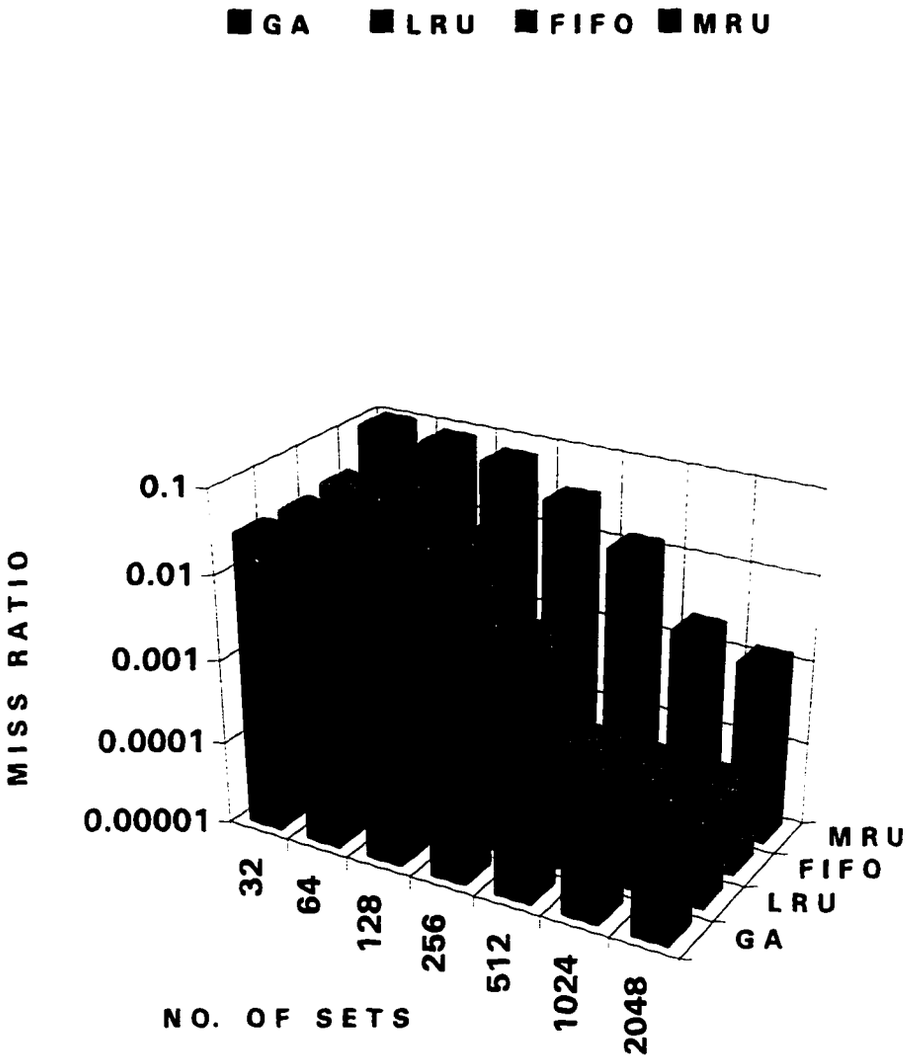


FIGURE 5.28: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH EAR AS THE TRACE FILE (LINE SIZE=16 AND ASSOCIATIVITY=4)

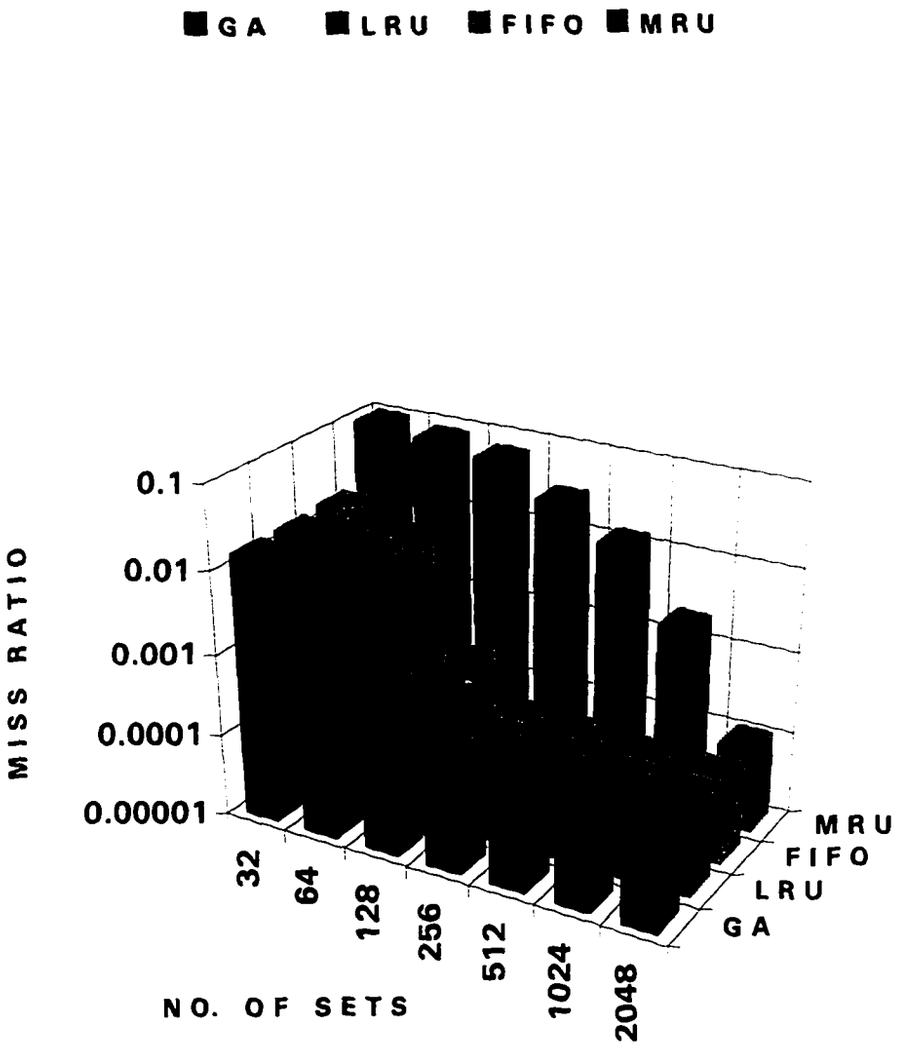


FIGURE 5.29: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND THE MRU ALGORITHMS WITH EAR AS THE TRACE FILE
(LINE SIZE = 16 AND ASSOCIATIVITY = 8)

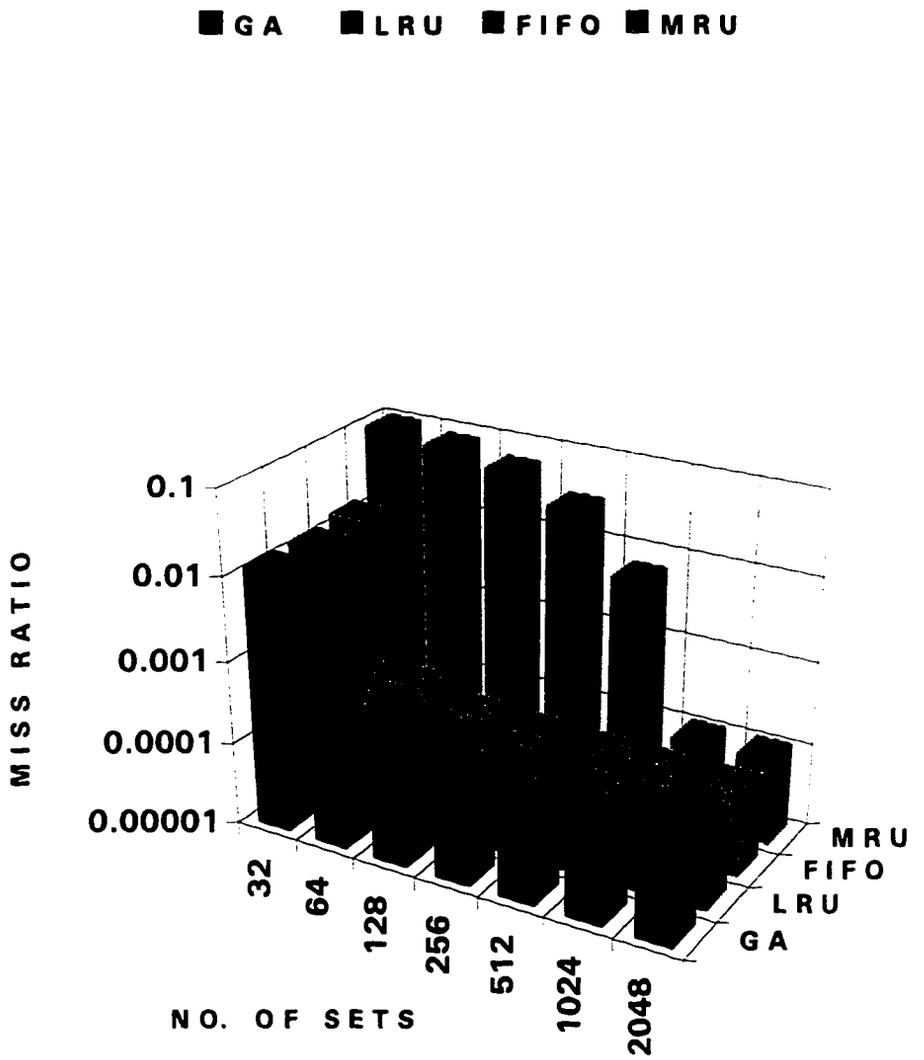


FIGURE 5.30: MISS RATIO vs. CACHE SIZE FOR THE GA, LRU, FIFO, AND MRU ALGORITHMS WITH EAR AS THE TRACE FILE (LINE SIZE = 16 AND ASSOCIATIVITY = 16)

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE RESEARCH

6.1 Summary and Conclusion

In this thesis, three new and improved cache replacement algorithms were presented. The proposed algorithms use neural networks for the purpose of guiding line replacement decision in caches. We began our studies by looking at the conventional line replacement strategies. In particular, simulation experiments were conducted to observe the relative behavior of the LRU and OPT replacement policies. The experiments indicated that it was indeed possible to develop near-optimal cache replacement algorithms. It was also observed that, on average, 75% of lines in the LRU and OPT stacks were identical, and about 25% of the LRU stack was made up of unique inactive lines. Moreover, the initial experiments provided clues for developing new and improved algorithms. Knowledge gained from the trace-driven simulations was used to understand the problem from a different perspective. The line replacement problem was, eventually, reformulated as the one related to the recognition of live and dead lines.

Neural networks were invoked, in chapter 3, to solve the reformulated problem. Cache algorithms were developed for different classes of neural networks. Performance evaluation studies of the algorithms were then conducted for TPC.DATA and BI.DATA benchmark trace files. Extensive trace driven simulation were performed on 21 different cache configurations encompassing the entire spectrum of practical cache organizations. Six popular neural network paradigms that were used during the experimentation are: Backpropagation Neural Network, Modular Neural Network, Radial Basis Functional Network, Learning Vector Quantization, Probabilistic Neural Network, and General Regression Neural Network. Our experiments revealed that Backpropagation neural network paradigm was very successful in achieving good performance. We also experimented with several variants of Backpropagation neural network model. It was observed that for TanH transfer function and Norm-Cum learning rule, Backpropagation neural network was able to achieve 16.47% improvement in the miss ratio over the LRU. This represents significant improvement in cache performance as it leads to considerable savings in the number of misses for a wide variety of systems and application programs.

Based on the clues provided by the initial experimentation, a generalized neural network based cache replacement algorithm was developed in chapter 5. The performance of algorithm when tested against the TPC.dat and BI.DATA trace files, was found to be good and consistent over the domain of simulated caches. The best performance for GA algorithm resulted in an improvement of 13.88% over the LRU. In order to consolidate the conclusions and verify that good, stable, and consistent behavior is exhibited by the GA algorithm, we evaluated its performance for four different SPEC benchmark trace files. Forty two different cache configurations were simulated and studied. The experiments indicated that the GA algorithm indeed outperformed the conventional algorithms. For the worst case, its performance was found to be same as the LRU.

6.2 Future Directions

Excellent performance of neural network based cache replacement algorithms means that this new approach has potential for providing promising results when applied to the page replacement and prefetching algorithms in virtual memory systems. The strategies presented here could be studied in a multiprocessor environment. Also, experiments may be conducted to see how the algorithms learn and perform with multi level caches. Cache designers and researchers are recommended to study and evaluate other neural network paradigms that could be associated with the algorithms presented in this thesis. Timing constraints, implementation aspects, and modifications to the algorithms are few other topics for research and development.

APPENDIX A

MEMORY CHARACTERISTICS OF A TYPICAL MAINFRAME COMPUTER WITH FOUR LEVEL MEMORY HIERARCHY

Memory level Characteristics	Level 0 CPU Registers	Level 1 Cache	Level 2 Main Memory	Level 3 Disk Storage	Level 4 Tape Storage
Device Technology	ECL	256-Kbit SRAM	4-Mbit DRAM	1-Gbyte Magnetic disk unit	5-Gbyte Magnetic tape unit
Access Time	10ns	25-40 ns	60-100 ns	12-20 ms	2-20 min
Capacity (in bytes)	512 bytes	128 Kbytes	512 Mbytes	60G-228G bytes	512G-2T bytes
Cost (in cents/KB)	18,000	72	5.6	0.23	0.01
Bandwidth (in MB/s)	400-800	250-400	80-133	3-5	0.18-0.23
Units of Transfer	4-8 bytes per word	32 bytes per block	0.5-1 Kbytes per page	5-512 Kbytes per file	Backup storage
Allocation Management	Compiler assignment	Hardware control	Operating system	Operating system/ User	Operating system/ User

APPENDIX B

LEGEND INFORMATION FOR BACKPROPAGATION NEURAL NETWORK FIGURES

T1: HIDDEN LAYER1=11, LEARNING RULE=QUICK-PROP, LEARNING RATE=0.5, TRANSFER FUNCTION=TanH, MOMENTUM=0.5
T2: HIDDEN LAYER1=11, LEARNING RULE=MAX-PROP, LEARNING RATE=0.5, TRANSFER FUNCTION=TanH, MOMENTUM=0.5
T3: HIDDEN LAYER1=19, LEARNING RULE=NORM-CUM, LEARNING RATE=0.5, TRANSFER FUNCTION=TanH, MOMENTUM=0.5
T4: HIDDEN LAYER1=11, LEARNING RULE=EXT-DELTA-BAR-DELTA, LEARNING RATE=0.5, TRANSFER FUNCTION=SIGMOID, MOMENTUM=0.5
T5: HIDDEN LAYER1=11, HIDDEN LAYER2=19, LEARNING RULE=MAX-PROP, LEARNING RATE=0.9, LEARNING RULE=TanH, MOMENTUM=0.9
T6: HIDDEN LAYER1=19, LEARNING RULE=NORM-CUM, LEARNING RATE=0.9, TRANSFER FUNCTION=TanH, MOMENTUM=0.9
T7: HIDDEN LAYER1=19, LEARNING RULE=NORM-CUM, LEARNING RATE=0.05, TRANSFER FUNCTION=TanH, MOMENTUM=0.05

APPENDIX C

DESCRIPTION OF SPEC BENCHMARK SUITES

SPEC CINT 92:

(1) 0.008.espresso
Language: C
Application: Logic Design
Description: Generates and optimizes Programmable Logic Arrays.

(2) 0.22.li
Language: C
Application: Interpreters
Description: Uses a LISP interpreter to solve the nine queens problem, using a recursive backtracking algorithm.

(3) 023.eqntott
Language: C
Application: Logic Design
Description: Translates a logical representation of a Boolean equation to a truth table.

(4) 026.compress
Language: C
Application: Data Compression
Description: Reduces the size of input files by using Lempel-Ziv coding.

(5) 072.sc
Language: C
Application: Spreadsheet
Description: Calculates budgets, SPEC metrics and amortization schedules in a spreadsheet based on the UNIX cursor-controller package,cursea.

(6) 095.gcc
Language: C
Application: Compiler
Description: Translates preprocessed C source files into optimized Sun-3 assembly language output

SPEC CFP 92:

(1) 013.spice2g6
Language: FORTRAN
Application: Circuit Design
Description: Simulates analog circuits (double precision).

(2) 015.doduc
Language: FORTRAN
Application: Simulation
Description: Performs Monte-Carlo simulation of the time evolution of a thermo-hydraulic model for a nuclear reactor's component (double precision).

(3) 034.mdljdp2
Language: FORTRAN
Application: Quantum Chemistry
Description: Solves motion equations for a model of 500 atoms interacting through the idealized Lennard-Jones potential (double precision).

(4) 039.wave5
Language: FORTRAN
Application: Electromagnetism
Description: Solves particle and Maxwell's equations on a Cartesian mesh (single precision).

(5) 047.tomcatv
Language: FORTRAN
Application: Geometric Translation
Description: Generates two-dimensional, boundary-fitted coordinate systems around general geometric domains (vectorizable, double precision).

(6) 048.ora
Language: FORTRAN
Application: Optics
Description: Traces rays through an optical surface containing spherical and planar surfaces (double precision).

(7) 052.alvinn
Language: C
Application: Robotics
Description: Trains a neural network using back propagation (single precision).

(8) 056.ear
Language: C
Application: Medical Simulation
Description: Simulates the human ear by converting a sound file to a cochleagram using Fast Fourier Transforms and other math library functions (single precision).

(9) 077.mdljsp2
Language: FORTRAN
Application: Quantum Chemistry
Description: Similar to 034.mdljdp2, solves motion equations for a model of 500 atoms (single precision).

(10) 078.swm256
Language: FORTRAN
Application: Simulation
Description: Solves the system of shallow water equations using finite difference approximations (single precision).

(11) 089.su2cor
Language: FORTRAN
Application: Quantum Physics
Description: Calculates masses of elementary particles in the framework of the Quark Gluon theory (vectorizable, double precision).

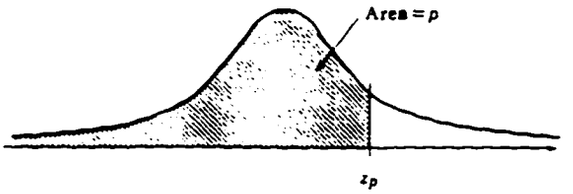
(12) 090.hydro2d
Language: FORTRAN
Application: Astrophysics
Description: Uses hydrodynamical Navier Stokes equations to calculate galactical jets (vectorizable, double precision)

(13) 093.nasa7
Language: FORTRAN
Application: NASA Kernels
Description: Executes seven program kernels representative of operations used frequently in NASA applications, such as Fourier transforms and matrix manipulations (double precision).

(14) 094.fpppp
Language: FORTRAN
Application: Quantum Chemistry
Description: Calculates multi-electron Integral derivatives (double precision).

APPENDIX D

QUANTILES OF THE UNIT NORMAL DISTRIBUTION



Quantiles of the Unit Normal Distribution

p	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
0.5	0.000	0.025	0.050	0.075	0.100	0.126	0.151	0.176	0.202	0.228
0.6	0.253	0.279	0.305	0.332	0.358	0.385	0.412	0.440	0.468	0.496
0.7	0.524	0.553	0.583	0.613	0.643	0.674	0.706	0.739	0.772	0.806
0.8	0.842	0.878	0.915	0.954	0.994	1.036	1.080	1.126	1.175	1.227

p	0.000	0.001	0.002	0.003	0.004	0.005	0.006	0.007	0.008	0.009
0.90	1.282	1.287	1.293	1.299	1.305	1.311	1.317	1.323	1.329	1.335
0.91	1.341	1.347	1.353	1.359	1.366	1.372	1.379	1.385	1.392	1.398
0.92	1.405	1.412	1.419	1.426	1.433	1.440	1.447	1.454	1.461	1.468
0.93	1.476	1.483	1.491	1.499	1.506	1.514	1.522	1.530	1.538	1.546
0.94	1.555	1.563	1.572	1.580	1.589	1.598	1.607	1.616	1.626	1.635
0.95	1.645	1.655	1.665	1.675	1.685	1.695	1.706	1.717	1.728	1.739
0.96	1.751	1.762	1.774	1.787	1.799	1.812	1.825	1.838	1.852	1.866
0.97	1.881	1.896	1.911	1.927	1.943	1.960	1.977	1.995	2.014	2.034
0.98	2.054	2.075	2.097	2.120	2.144	2.170	2.197	2.226	2.257	2.290

p	0.0000	0.0001	0.0002	0.0003	0.0004	0.0005	0.0006	0.0007	0.0008	0.0009
0.990	2.326	2.330	2.334	2.338	2.342	2.346	2.349	2.353	2.357	2.362
0.991	2.366	2.370	2.374	2.378	2.382	2.387	2.391	2.395	2.400	2.404
0.992	2.409	2.414	2.418	2.423	2.428	2.432	2.437	2.442	2.447	2.452
0.993	2.457	2.462	2.468	2.473	2.478	2.484	2.489	2.495	2.501	2.506
0.994	2.512	2.518	2.524	2.530	2.536	2.543	2.549	2.556	2.562	2.569
0.995	2.576	2.583	2.590	2.597	2.605	2.612	2.620	2.628	2.636	2.644
0.996	2.652	2.661	2.669	2.678	2.687	2.697	2.706	2.716	2.727	2.737
0.997	2.748	2.759	2.770	2.782	2.794	2.807	2.820	2.834	2.848	2.863
0.998	2.878	2.894	2.911	2.929	2.948	2.968	2.989	3.011	3.036	3.062
0.999	3.090	3.121	3.156	3.195	3.239	3.291	3.353	3.432	3.540	3.719

BIBLIOGRAPHY

[1] L.Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer, "IBM Systems Journal, 5(2), pp.78-101, 1996.

[2] L.Belady and C.J.Kuehner, "Dynamic Space Sharing in Computer Systems, "CACM, 12(5), pp.282-288, May, 1969.

[3] R.Mattson, J.Gecsei, D.Slutz and, I.Traiger, "Evaluation Techniques for Storage Hierarchies, "IBM Systems Journal, 9(2), pp.78-117, 1970.

[4] M.S.Obaidat, H.Khalid and, K.Sadiq, "A Methodology For Evaluating The Performance of CISC Computer Systems Under Single And Two-Level Cache Environments, "Microprocessing and Microprogramming Journal: Elsevier Publisher, 40(6), pp.411-421, July, 1994.

[5] S.Laha, J.H.Patel and, R.K.Iyer, "Accurate Low-Cost Methods For Performance Evaluation of Cache Memory Systems, "IEEE Trans. on Computers, 37(11), pp.1325-1336, Nov., 1988.

[6] M.S.Obaidat, H.Khalid and, K.Sadiq, "Performance Evaluation of CISC Computer Systems Under Single And Two-Level Cache Environments, "Proceedings of The 22nd Annual ACM Computer Science Conference, pp.260-268, March, 1994.

[7] H.Khalid and M.S.Obaidat, "A Novel Cache Memory Controller: Algorithm And Simulation, "Proceedings of The 1995 Summer Computer Simulation Conference (SCSC'95), Canada, pp.767-772, July 24-25, 1995.

[8] P.M.Fenwich, "Some Aspects of The Dynamic Behavior of Hierarchical Memories, "IEEE Trans. on Computers, 34(6), pp.570-573, June, 1985.

[9] M.S.Obaidat and H.Khalid, "A Performance Evaluation Methodology for Computer Systems, "1995 IEEE 14th Annual International Phoenix Conference on Computers and Communications (IPCCC'95), pp.713-719, March, 1995.

[10] H.Khalid and M.S.Obaidat, "Near-Optimal Cache Replacement Policy For High Performance Computer Systems, "Submitted to the IEEE Trans. on Computers,

1995.

[11] K.Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programmability, Computer Engineering Series, McGraw-Hill, 1993.

[12] H.Khalid and M.S.Obaidat, "High Performance Cache Memory Replacement Schemes, "1995 IEEE Annual International Conference on Electronics, Circuits and, Systems (ICECS'95), pp.1-7, Dec., 1995.

[13] H.Khalid and M.S.Obaidat, "Performance Evaluation of a New Cache Replacement Scheme Using SPEC, "Proceedings of the 1996 IEEE International Phoenix Conference on Computers and Communication (IPCCC'96), pp.144-150, March 27-29, 1996.

[14] W.W.Hwu and P.P.Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler, "Proceedings of 16th Annual International Symposium on Computer Architecture, pp.242-251, June, 1989.

[15] D.Thiébaut, H.S.Stone and, J.L.Wolf, "Improving Disk Cache Hit-Ratios Through Cache Partitioning, "IEEE Trans. on Computers, 41(6), pp.665-676, June, 1992.

[16] A.Agarwal, M.Horowitz and, J.Hennessy, "An Analytical Cache Model, "ACM Trans. on Computer Systems, 7(2), pp.184-215, May, 1989.

[17] J.E. Smith and J.R. Goodman, "Instruction Cache Replacement Policies and Organizations, "IEEE Trans. On Computers, 34(3), pp.234-241, March, 1985.

[18] M.D. Hill and A.J. Smith, "Evaluating Associativity in CPU Caches, "IEEE Trans. on Computers, 38(12), pp.1612-1630, Dec., 1989.

[19] S.G. Tucker, "The IBM 3090 Systems: An Overview, "IBM Systems Journal, 25(6), Jan., 1986.

[20] P.J.Denning, "On Modeling Program Behavior, "Proceedings of SJCC, pp. 937-944, 1972.

[21] H.S.Stone, High Performance Computer Architecture, Addison-Wesley Publishing Company, Reading, MA, 1993.

[22] A.J. Smith, "Design of CPU Cache Memories, "Proceedings of IEEE TENCON'87, Vol.3, pp.(30.2.1-30.2.10), August, 1987.

[23] C.E.Wu, Y.Hsu and, Y.-H.Liu, "A Quantitative Evaluation of Cache Types for High Performance Computer Systems, "IEEE Trans. on Computers, 42(10), pp.1154-1162, October, 1993.

[24] C.J.Conti, "Concepts For Buffer Storage, "IEEE Comp. Group News, 2(8), pp.9-13, March, 1969.

[25] J.Pomerene, T.R.Puzak, R.Rechtschaffen and, F.Sporacio, "Prefetching Mechanism For a High-Speed Buffer Store, "US Patent, 1984.

[26] W.Y.Chen, P.P.Chung, T.M.Conte, and W.-M.W.Hwu, "The Effect of Code Expanding Optimizations on Instruction Cache Design, "IEEE Trans. on Computers, 42(9), pp.1045-1057, Sept., 1993.

[27] S.Hiremath and M.A.Manzoul, "An Improved Fuzzy Replacement Algorithm for Cache Memories, "Cybernetics and Systems: An International Journal, Vol. 24, pp.325-339, 1993.

[28] M.J.Flynn, Computer Architecture: Pipelined and Parallel Processor Design, Jones and Barlett Publishers International, England, 1995.

[29] M.D.Hill, Aspects of Cache Memory and Instruction Buffer Performance, Ph.D. Thesis, Univ. of California-Berkeley, Nov., 1987.

[30] M.Kohayoshi and M.H.MacDougall, "The Stack Growth Function: Cache Line Reference Models, "IEEE Transactions on Computers, 38(6), pp.798-805, June, 1989.

[31] T.R.Puzak, Analysis of Cache Replacement Algorithms, Ph.D. Thesis, Univ. of Massachussetts, February, 1985.

[32] A.J.Smith, "Line (Block) Size Choice for CPU Cache Memories, "IEEE Trans. on Computers, 36(9),

pp.1063-1075, September, 1987.

[33] D.F.Thiebaut and H.S.Stone, "Footprints in the Cache, "ACM Transactions on Computer Systems, 5(4), pp.305-329, Nov., 1987.

[34] W.S.Wong and R.J.T.Morris, "Benchmark Synthesis Using LRU Cache Hit Function, "IEEE Transactions on Computers, 37(6), pp.637-645, June, 1988.

[35] M.S.Obaidat, "A Trace Driven Simulation Study of Two Level Cache System, "Journal of Computer and Electrical Engineering, 21(3), pp.201-210, May, 1995.

[36] A.Agarwal, R.Sites, M.Horowitz, "ATUM: A New Technique for Capturing Addresss Traces Using Microcode, "Proceedings of the 13th Annual International Symposium on Computer Architecture, pp.119-129, Japan, June, 1986.

[37] A.Agarwal, P.Chow, M.Hrowitz, J.Acken, A.Salz, J.Hennessy, "On-Chip Instruction Caches for High Performance Processors, "In Advanced Research in VLSI, pp.1-24, Standford University, March, 1987.

[38] A.Agarwal, Analysis of Cache Performance for Operating Systems and Multiprogramming, Ph.D. Thesis, Standford University, Available as Technical Report CSI Tr-87-332, May 1987.

[39] A.Agarwal, J.Hennessy, M.Horowitz, "Cache Performance of Operating System and Multiprogramming workloads, "ACM Transaction on Computer Systems, 7(2), May, 1989.

[40] K.Akeley, "The Silicon Graphics 4D/240GTX Superworkstation, "IEEE Computer Graphics and Applications, 9(4), pp.71-84, July, 1989.

[41] C.Alexander, W.Keshlear, E.Cooper, F.Briggs, "Cache Memory Performance in a Unix Enviornment, " SigArch News, 14(3), pp.41-70, June, 1986.

[42] D.Alpert, "Performance Tradeoffs for Microprocessor Cache Memomries, "Technical Report CSI-TR-83-239, Standford University, December, 1983.

[43] D.Alpert, Memory Hierarchies for Directly Executed Language Microprocessors, Ph.D. Thesis, Standford University, Available as Technical Report CSL-TR-84-260, 1984.

[44] D.Alpert, M.Flynn, "Performance Tradeoffs for Microprocessor Cache Memories, "IEEE Micro, 8(4), pp.44-55, August, 1988.

[45] Am29000 User's Manual, Advanced Micro Devices, Inc., Sunnydale, CA., 1987.

[46] D.W.Archer, D.R.Deverek, T.F.Fox, P.E.Gronowski, R.K.Jain, M.Leary, D.G.Miner, A.Olesin, S.D.Persels, P.I.Rubinfeld, R.M.Supnik, "A CMOS VAX Microprocessor with On-Chip Cache and Memory Management, "Journal of Solid State Circuits, SC-22(5), pp.849-852, October, 1987.

[47] J.Archibald, J.L.Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model, "ACM Transactions on Computer Systems, 4(4), pp.273-298, November, 1986.

[48] R.R.Atkinson, E.M.McCreight, "The Dragon Processor, "Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems, "ASPLOS-II, pp.65-71, October, 1987.

[49] J.L.Baer, W.H.Wang, Architectural Choices for Multi-Level Cache Hierarchies, Technical Report TR-87-01-04, Department of Computer Science, University of Washington, January, 1987.

[50] J.L.Baer, W.H.Wang, "On the Inclusion Properties for Multi-Level Cache Hierarchies, "Technical Report Tr-87-11-08, Department of Computer Science, University of Washington, November, 1987.

[51] R.Bellman, Dynamic Programming, Princeton University Press, Princeton, NJ, 1957.

[52] B.T.Bennett, J.H.Pomerene, T.R.Puzak, R.N.Rechtschaffen, "Prefetching in a Multilevel Hierarchy, "IBM Technical Disclosure Bulletin, 25(1), pp.88-89, June, 1982.

[53] A.D.Berenbaum, B.W.Colbry, D.R.Ditzel, H.R.Freeman, H.R.McLellan, K.J.O'Connor, M.Shoji, "CRISP: A Pipeline 32-Bit Microprocessor with 13-Kbit of Cache Memory, "Solid State Circuits Journal, SC-22(5), pp.776-782, October, 1987.

[54] A.Borg, R.E.Kessler, G.Lazana, D.W.Wall, Long Address Traces From RISC Machines: Generation And Analysis, WRL Research Report 89/14, Western Research Laboratory, Digital Equipment Corp., 1989.

[55] J.Cho, A.J.Smith, H.Sachs, The Memory Architecture and the Cache and Memory Management Unit for the Fairchild CLIPPEr Processor, Technical Report UCB/CSD 86/289, Computer Science Division, University of California, Berkeley, April, 1986.

[56] C.K.Chow, "Determining the Optimum Capacity of a Cache Memory, "IBM Technical Disclosure Bulletin, 17(10), pp.3163-3166, March, 1975.

[57] C.K.Chow, "Determination of Cache's Capacity and its Matching Storage Hierarchy, "IEEE Transaction on Computers, C-25(2), pp.157-164, February, 1976.

[58] D.W.Clark, "Cache Performance in the VAX-11/780, "ACM Transaction on Computer Systems, 1(1), pp.24-37, February, 1983.

[59] D.W.Clark, "Pipelining and Performance of the VAX 8800 Processor, "Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II), pp.173-178, IEEE, October, 1987.

[60] E.I.Cohen, G.M.King, J.T.Brady, "Storage Hierarchies, "IBM Systems Journal, 28(1), pp.62-76, 1989.

[61] C.B.Cole, "Advanced Cache Chips Make the 32-Bit Microprocessors Fly, "Electronics, 60(13), pp.78-79, June 11, 1988.

[62] D.J.Colglazier, "A performance Analysis of Multiprocessors Using Two-Level Caches, "Technical Report CSG-36, University of Illinois at Urbana Champaign, August, 1984.

[63] VAX Hardware Handbook, Digital Equipment Corp., Maynard, MA, 1980.

[64] P.J.Denning, "The Working Set Model for Program Behavior, "Communications of the ACM, 11(5), pp.323-333, May, 1968.

[65] R.R.Duncombe, The SPUR Instruction Unit: An On-Chips Instruction Cache Memory for a High Performance VLSI Multiprocessor, Technical Report UCB/CSD 87/307, University of California, Berkeley, August, 1986.

[66] M.Easton and R.Fagin, "Cold Start vs. Warm Start Miss Ratios, "Communications of the ACM, 21(10), pp.866-872, October, 1978.

[67] R.W.Edenfield, M.G.Gallup, W.B.Ledbetter Jr., R.C.Mcgarity, E.E.Qunitana, R.A.Reininger, "The 68040 Processor: Part I, Design and Implementation, "IEEE Micro, 10(1), pp.66-79, February, 1990.

[68] S.Egger, R.Katz, "Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation, "Proceedings of the 15th Annual

International Symposium on Computer Architecture, pp.373-383, June, 1988.

[69] D.Freitas, "32-Bit Processor Achieves Sustained Performance of 20 MIPS, "Proceedings of Northcom, October, 1988.

[70] J.Fu, J.B.Keller, K.J.Haduch, "Aspects of the VAX 8800C Box Design, "Digital Technical Journal, 1(4), pp.41-51, February, 1987.

[71] R.Garner, A.Agarwal, E.W.Briggs, D.Brown, M.N.Joy, S.Klienman, S.Muchnick, M.Namjoo, D.Patterson, J.Pendleton, K.G.Tan, R.Tuck, "The Scalable Processor Architecture (SPARC), "In Digest of Papers, COMPCON 88, pp.2-14, February, 1988.

[72] J.Gecsei, "Determining Hit Ratios for Multilevel Hierarchies, "IBM Journal of Research and Development, 18(4), pp.316-327, July, 1974.

[73] B.S.Gindele, "Buffer Block Prefetching Method, "IBM Technical Disclosure Bulletin, 20(2), pp.696-697, July, 1977.

[74] J.R.Goodman, "Cache Memory Optimization to Reduce Processor/Memroy Traffic, "Department of Computer Science, University of Wisconsin, Madison, 1985.

[75] J.R.Goodman, "Coherency for Multiprocessor Virtual Address Caches, "Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II), pp.72-81, October, 1987.

[76] G.Gygax, Advance Dungeons and Dragons Players Handbook, TSR Games, Inc., Lake Geneva, WI, Page 96, 1978.

[78] I.J.Haikala, P.H.Kutvonen, "Split Cache Organizations, "Performance '84, pp.459-472, 1984.

[79] I.J.Haikala, Program Behavior in Memory Hierarchies, Ph.D. Thesis, University of Helsinki, Available as Technical Report A-1986-2, 1986.

[80] V.C.Hamacher, Z.G.Vranesic, S.G.Zaky, Computer Organization, McGraw Hill, New York, NY, 1978.

[81] A.Hattori, M.Koshino, S.Kamimoto, "Three-level Hierarchical Storage System for the FACOM M-380/382, "Proceedings of Information Processing IFIP, pp.639-697, 1983.

[82] J.L.Hennessy, D.A.Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, San Mateo, CA, 1990.

[83] M.D.Hill, A.J.Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories, "Proceedings of the 11th Annual International Sysmposium on Computer Architecture, pp.158-166, June, 1984.

[84] M.D.Hill, S.J.Eggers, J.R.Larus, G.S.Taylor, G.Adams, B.K.Bose, G.A.Gibson, P.M.Hansen, J.Keller, S.I.Kong, D.Lee, J.M.Pendleton, S.A.Ritchie, D.A.Wood, "SPUR: A VLSI Multiprocessor Workstation, "Technical Report UCB/CSD 86/273, Computer Science Division, University of California, Berkeley, December, 1985.

[85] M.D.Hill, S.J.Eggers, J.R.Larus, G.S.Taylor, G.Adams, B.K.Bose, G.A.Gibson, P.M.Hansen, J.Keller, S.I.Kong, D.Lee, J.M.Pendleton, S.A.Ritchie, D.A.Wood, B.G.Zorn, P.N.Hilfinger, D.Hodges, R.H.Katz, J.Ousterhout, D.A.Patterson, "Design Decisions in SPUR, "IEEE Computer, 19(11), pp.8-24, November, 1986.

[86] M.D.Hill, "The Cache for Direct-Mapped Caches, " IEEE Computer, 21(12), pp.25-41, December, 1988.

[87] G.Hinton, R.Riches, C.Jasper, K.Lai, "A Register Scoreboarding Mechanism, "IEEE International Solid-State Circuits Conference, Digest of Technical Papers, pp.270-271, February, 1988.

[88] M.Horowitz, P.Chow, D.Stark, R.Simoni, A.Salz, S.Przybylski, J.Hennessy, G.Gulak, A.Agrwal, J.Acken, "MIPS-X: A 20 MIPS Peak, 32-Bit Microprocessor with On-Chip Cache, "Solid State Circuits Journal, SC-22(5), pp.790-799, October, 1987.

[89] R.Jog, G.S.Sohi, M.K.Vernon, "The TREEBus Architecture and Its Analysis, "Technical Report CS 747, Computer Science Department, University of

Wisconsin- Madison, February, 1988.

[90] N.Jouppi, J.Dion, D.Boggs, M.J.K.Nielsen, "MutiTitan: Four Architecture Papers, "WRL Research Report Laboratory, Digital Equipment Corp., April, 1988.

[91] N.P.Jouppi, "Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU, "Proceedings of the 14th Annual International Symposium on Computer Architecture, pp.281-290, May, 1989.

[92] S.R.Karlovsky, "Automatic Management of Programmable Caches: Algorithms and Experience, "CSRD Report 892, Center of Supercomputing Research and Development, University of Illinois at Urbana - Champaign, July, 1989.

[93] R.H.Katz, S.J.Eggers, G.A.Gibson, P.M.Hansen, M.D.Hill, J.M.Pendleton, S.A.Ritchie, G.S.Taylor, D.A.Wood, D.A.Patterson, "Memory Hierarchy Aspects of a Multiprocessor RISC: Cache and Bus Analyses, "Technical Report USB/CSD 85/221, Computer Science Div., Univ. of California, Berkeley, January, 1985.

[94] E.Killian, pixie(1): UMIPS Man Page, MIPS Computer Systems, Sunnyvale, CA, 1986.

[95] E.Killian, "Miss Ratio Data as Function of Cache and Block Sizes, "Personal Communication, MIPS Computer Systems, Sunnyvale, CA, January, 1988.

[96] L.Kohn, S.-W.Fu, "A 1,000,000 Transistor Microprocessor, "IEEE International Solid-State Circuits Conferenece, Digest of Technical Papers, pp.54-55, February, 1989.

[97] S.Laha, J.H.Patel, R.K.Iyer, "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems, "IEEE Transactions on Computers, 37(11), pp.1325-1336, November, 1988.

[98] L.L.Lapin, Probability and Statistics for Modern Engineering, PWS Publishers, Boston, MA, 1983.

[99] R.E.Larson, J.L.Casti, Principles of Dynamic Programming, Marcel Dekker, Inc., New York, NY, 1978.

[100] R.L.Lee, P.-C.Yew, D.H.Lawrie, "Data Prefetching in Shared Memory Multiprocessors, "CSRD Report 639, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, January, 1987.

[101] J. S. Liptay, "Structural Aspects of the System/360 Model 85, Part II: The Cache, "IBM Systems Journal, 7(1), pp.15-21, 1968.

[102] J.E.MacDonald, K.L.Sigworth, "Storage Hierarchy Optimization Procedure, "IBM Journal of Research and Development, 19(2), pp.164-170, March, 1975.

[103] D.MacGregor, D.Mothersole, B.Moyer, "The Motorola MC68020, "IEEE Micro, 4(8), pp.101-118, August, 1984.

[104] M.J.Mahon, R.B.-L.Lee, J.C.Huck, W.R.Bryg, " Hewlett-Packard Precision Architecture: The Processor, "Hewlett-Packard Journal, 37(8), pp.4-22, August, 1986.

[105] T.Manuel, "Taking a Close Look at the Motorola 88000, "Electronics, 61(9), pp.75-78, April 11, 1988.

[106] J.R.Mashey, "MIPS and the Motion of Complexity, "Uniform Conference Proceedings, 1986.

[107] R.E.Matick, Computer Storage Systems and Technology, John Wiley and Sons, New York, NY, 1977.

[108] R.L.Mattson, J.Gecsei, D.R.Slutz, I.L.Traiger, " Evaluation Techniques for Storage Hierarchies, "IBM Systems Journal, 9(2), pp.78-117, 1970.

[109] C.McCrosky, "An Analytical Model of Cache Memories, "Technical Report 86-7, Department of Computational Science, University of Saskatchenwan, 1986.

[110] S.McFarling, "Program Optimization for Instruction Caches, "Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III), pp.183-191, IEEE, April, 1989.

[111] MIPS Performance Brief, MIPS Computer Systems, Sunnyvale, CA, October, 1987.

[112] C.Mitchell, Architecture and Cache Simulation Result for Individual Benchmarks, Ph.D. Thesis, Standford University, Available as Technical Report CSL-TR-86-296, 1986.

[113] E.N.Miya, "Multiprocessor/Distributed Processing Bibliography, "SigArch News, 13(1), pp.27-29, March, 1985.

[114] J.Moussouris, L.Crudele, D.Freitas, C.Hansen, E.Hudson, R.March, S.Przybylski, T.Riordan, C.Rowen, D.Van't Hof, "A CMOS RISC Processor with Integrated System Functions, "Digest of Papers, COMPCON 86, pp.126-131, March, 1986.

[115] J.M.Mulder, Tradeoffs in Processor-Architecture and Data-Buffer Design, Ph.D. Thesis, Standford University, Available as Technical Report CSL-TR-87-345, December, 1987.

[116] C.-J.Peng, G.S.Sohi, "Cache Memory Design Consideration to Support Languages with Dynamic Heapo Allocation, Technical Report 860, Computer Sciences Department, University of Wisconsin-Madison, July, 1989.

[117] A. V. Pohm and O. P. Agrawal, High-Speed Memory Systems, Reston Publishing Company, 1983.

[118] S.Przybylski, Performance-Directed Memory Hierarchy Design, Ph.D.Thesis, Standford University, Available as Technical Report CSL-TR-88-366, September, 1988.

[119] G.S.Rao, Performance Analysis of Cache Memories, "Journal of the ACM, 25(3), pp.378-395, July, 1978.

[120] B.R.Rau, G.Rossman, "The Effect of Instruction Fetch Strategies Upon the Performance of Pipelined Instructions Units, "Proceedings of the 14th Annual International Symposium on Computer Architecture, pp.80-89, June, 1977.

[121] B.R.Rau, Program Behavior and the Performance of Memory Systems, Ph.D. Thesis, Standford University, 1977.

[122] B.R.Rau, "Sequential Prefetch Strategies for Instructions and Data, "Technical Report CSL-TR-77-131, Digital Systems Laboratory, Standford University, January, 1977.

[123] D.Roberts, T.Layman, G.Taylor, "An ECL RISC Microprocessor Designed for Two level Cache, "Digest of Papers, COMPCON 90, pp.228-231, February, 1987.

[124] R.T.Short, A Simulation Study of Multileve Cache Memories, Master's Thesis, Department of Computer Science, University of Washington, January, 1987.

[125] R.T.Short, H.M.Levy, "A Simulation Study of Two-Level Caches, "Proceedings of the 15th Annual International Symposium on Computer Architecture, pp.81-89, June, 1988.

[126] R.R.Sieworek, C.G.Bell, A.Newell, Computer Structures Principle and Examples, McGraw Hill, NewYork, NY, 1982.

[127] J.P.Singh, H.S.Stone, D.F.Thiebaut, "An Analytical Model for Fully Associative Cache Memories, "Research Report RC 14232(#63678), IBM, November, 1988.

[128] A.J.Smith, "Two Methods for the Efficient Analysis of Memory Address Trace Data, "IEEE Transactions on Software Engineering, SE-3(1), January, 1977.

[129] A.J.Smith, "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory, "IEEE Transactions on Software Engineering, SE-4(2), pp.121-130, March, 1978.

[130] A.J.Smith, "Sequential Program Prefetching in Memory Hierarchies, "IEEE Computer, 11(12), pp.7-21, December, 1978.

[131] A.J.Smith, "Sequentiality and Prefetching in Database Systems, "ACM Transactions on Database Systems, 3(3), pp.223-247, Septmber, 1978.

[132] A.J.Smith, "Characterizing the Storage Process and Its Effects on Main Memory Update, "Journal of the ACM, 26(1), pp.6-27, January, 1979.

[133] A.J.Smith, "Cache Memories, "ACM Computing Surveys, 14(3), pp.473-530, September, 1982.

[134] J.E.Smith, J.R.Goodman, "A Study of Instruction Cache Organizations and Replacement Policies, "Proceedings of the 10th Annual International Symposium on Computer Architecture, pp.132-137, June, 1983.

[135] A.J.Smith, "Cache Evaluation and the Impact of Workload Choice, "Proceedings of the 12th Annual International Symposium on Computer Architecture, pp.64-73, June, 1985.

[136] A.J.Smith, "Problems, Directions and Issues in Memory Hierarchies, "Proceedings of the 18th Annual Hawaii Conference on System Sciences, pp.468-476, 1985.

[137] A.J.Smith, "Bibliography and Readings on CPU Cache Memories and Related Topics, "Computer Architecture News, 14(1), pp.22-42, January, 1986.

[138] A.J.Smith, "Design of CPU Cache Memories, " Technical Report UCB/CSD 87/357, Computer Science Division, University of California, Berkeley, June, 1987.

[139] K.So, R.N.Rechtschaffen, "Cache Operations by MRU Changes, "IEEE Transactions on Computers, 37(6), pp.700-709, June, 1988.

[140] G.S.Sohi, M.-C.Chiang, Memory Organization for Multiprocessors with Onchip Cache Meomories, Unpublished Memo., Computer Sciences Department, University of Wisconsin - Madison, January, 1987.

[141] F.J.Sparacio, "Data Processing System with Second Level Cache, "IBM Technical Disclosure Bulletin , 21(6), pp.2468-2469, November, 178.

[142] H.Khalid and M.S.Obaidat, "KORA: A New Cache Replacement Scheme, "Submitted for publication in

Journal of Systems Architecture, 1996.

[143] H.S.Stone, High Performance Computer Architecture, Addison-Wesley, Reading, MA, 1990.

[144] W.D.Strecker, "Transient Behvaior of Cache Memories, "ACM Transactions on Computer Systems, 1(4), pp.281-293, November, 1983.

[145] D.Tanksalvala, J.Lamb, M.Buckley, B.Long, S.Chapin, J.Lotz, E.Delon, R.Luebs, K.Erskine, S.MuMullen, M.Forsyth, R.Novak, T.Gaddeis, D.Quarnstrom, C.Gleason, E.Rashid, D.Halperin, N.Sigal, H.Hill, C.Simpson, D.Hollenbeck, J.Spencer, R.Horning, H.Tran, T.Hotchkiss, D.Weir, D.Kipp, J.Wheeler, P.Knebel, J.Yeter, C.Kohlhardt, "A 90MHZ CMOS RISC CPU Designed for Sustained Performance, "IEEE International Solid-State Cirucuits Conference, Digest of Technical Papers, pp.52-53, February, 1990.

[146] C.P.Thacker, "Cache Strategies for Shared-Memory Multiprocessors, "New Frontiers in Computer Architecture, pp.51-62, February, 1990.

[147] C.P.Thacker, L.C.Stewart, "Firefly: A Multiprocessor Workstation, "Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II),pp.164-172, October 1987.

[148] D.F.Thiebaut, H.S.Stone, J.L.Wolf, "A Theory of Cache Behavior, "Research Report RC 13309, IBM, November, 1987.

[149] D.F.Thiebaut, H.S.Sone, J.L.Wolf, Synthetic Traces for Trace-Driven Simulation of Cache Memories, Research Report RC 14268(#63748), IBM, December, 1988.

[150] D.F.Thiebaut, "On the Fractal Dimension of Computer Programs and Its Applications to the Prediction of the Cache Miss Ratio, "IEEE Transactions on Computers, 38(7), pp.1012-1027, July, 1989.

[151] J.G.Thompson, "Efficient Analysis of Caching Systems, "Technical Report UCB/CSD 87/374, Computer Science Division, University of California, Berkeley, October, 1987.

[152] J.G.Thompson, A.J.Smith, "Efficient (Stack) Algorithms for Analysis of Write-Back and Sector Memories, "ACM Transaction on Computer Systems, 7(1), pp.78-116, February, 1989.

[153] ALS/AS Logic Data Book, Texas Instruments, Dallas, TX, 1986.

[154] R.M.Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units, "IBM Journal of Research and Development, 11(1), pp.25-33, January, 1967.

[155] M.K.Vernon, R.Jog, G.S.Sohi, "Performance Analysis of Hierarchical Cache-Consistent Multiprocessors, "Performance Evaluation, 9(4), pp.287-302, August, 1989.

[156] W.-H.Wang, J.-L.Baer, Levy H.M., "Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy, "Technical Report 88-11-02, Department of Computer Science, University of Washington, November, 1988.

[157] T.A.Welch, "Memory Hierarchy Configuration Analysis, "IEEE Transactions on Computers, C-27(5), pp.408-413, May, 1978.

[158] N.Wilhelm, Cache Design for the Titan Processor, Personal Communication, Western Research Laboratory, Digital Equipment Corp., October, 1987.

[159] J.M.Wilkes, "Slave Memories and Dynamic Storage Allocation, "IEEE Transactions on Electronic Computers, EC-14(2), pp.207-271, April, 1965.

[160] A.W.Wilson Jr., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors, "Proceedings of the 14th Annual International Symposium on Computer Architecture, pp.244-252, June, 1987.

[161] A.J.Aho, Hoperoft and, J. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley: Readings, Mass, 1974.

[162] P.J.Denning, "On Modeling Program Behavior, "Proc. Sprint Joint Computer Conference, AFIPS: Montvale, New Jersey, Vol.40, pp.937-944, 1970.

[163] P.J.Denning and S. Schwartz, "Properties of the Working Set Model, "CACM, 15(3), pp. 191-198, March, 1972.

[164] P.J.Denning, "Working Sets Past and Present, " IEEE Trans. on Software Engineering, SE-6(1), pp.219-228, Jan., 1980.

[165] W.J.Dixon and F.J. Massey, Introduction to Statistical Analysis, McGraw-Hill, New York, 1969.

[166] G.C.Driscoll, R. Matick, T. Puzak, and J. Shedlestsky, "Split With Variable Interleave Boundry, "IBM Tech Disclosure Bull., 22(11), pp.5183-5186, April, 1980.

[167] G.C.Driscoll, L. Hoevel, T. Puzak, and J. Shedletsky, "Design to improve cache performance via overlapping of cache miss sequences, "IBM Tech Disclousre Bull., 25(11b), pp.5962-5977, April, 1983.

[168] L.Hoevel and J. Voldman, "Mechanism for cache replacement and prefetching driven by heuristic estimation of operating system behavior, "IBM Tech

Disclosure Bull., 23(8), pp.3923- , Jan. 1981.

[169] L.Hoevel, J. Voldman, "Cache-line reclamation and cast-out avoidance under operating system control, "IBM Tech. Disclosure Bull., 23(8), pp. 3912- , Jan., 1981.

[170] M.Hofri and P. Tzelnic, "The Working Set Size Distribution for the Markov Chain Model of Program Behavior, "SIAM J. Computing, 11(3), pp.453-465, August, 1982.

[171] J.Pomerene, T. Puzak, R. Rechtschaffen., and F. Sparacio, "Improving L2 Performance in a Multi-Level Memory Hierarchy, "IBM Tech. Disclosure Bull., 25(8), pp.4203-, Jan., 1983.

[172] J.Pomerene, T. Puzak, R. Rechtschaffen., and F. Sparacio, "Enhanced Concurrency Using a Multi-Level Cache System, "IBM Tech. Disclosure Bull., 25(8), pp.5559-, Jan., 1983.

[173] J.Pomerene, T. Puzak, R. Rechtschaffen and, F. Sparacio, Prefetching Mechanism for a High Speed Buffer Store, Patent Pending.

[174] A.J.Smith, "A Modified Working Set Paging Algorithm, "IEEE Trans On Computing, pp.907-914, Sept. 1976.

[175] J.R.Spirn, Program Behavior Models and Measurements, Elsevier North-Holland: 1977.

[176] A.Agarwal, R.Simoni, M.Horowitx and, J.Hennessy, "An Evaluation of Directory Schemes for Cache Coherence, "Proc. of the 15th International Symposium on Computer Architecture, 1988.

[177] J.Gee, M.Hill, D.Pnevmatikatos and, A.Smith, "Cache Performance of the SPEC92 Benchmark Suite, "IEEE Micro, 13(4), pp.17-27, August 1993.

[178] J.Gee and A.J.Smith, "The Performance Impact of Vector Processor Caches, "Proc. of 25th Hawaii International Conference on System Sciences, Vol.1, pp.437-448, Jan., 1992.

[179] D.B.Glasco, B.A.Delagi and, M.J.Flynn, "Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessor, "Proc. of the 27th Annual Hawaii International Conference on System Sciences, Vol.1, pp.534-545, Jan., 1994.

[180] J.R.Goodman, "Cache Consistency and Sequential Consistency, "Technical Report 61, IEEE SCI Committee, March 1989.

[181] J.R.Goodman and W.-C. Hsu, "On the use of Registers vs. Cache to Minimize Memory Traffic, "Proc. of the 13th Annual International Symp. on Computer Architecture, pp.375-383, 1986.

[182] E.Hagersten, A.Landin and, S.Haridi, "DDM--A Cache-Only memeory Architecture, "IEEE Computer, 25(9), pp.44-54, Sept. 1992.

[183] D.Hammerstrom, Analysis of memory Addressing Architecture, Ph.D. Thesis, University of Illinois, July 1977.

[184] W.L. Lynch, The Interaction of Virtual Memory and Cache Memory, Ph.D. Thesis, Stanford University, CSL-TR-93-587, Oct.1993.

[185] C.Mitchell, Processor Architecture and Cache Performance, Ph.D. Thesis, Stanford University, CSL-TR-86-296, June 1986.

[186] J.M.Mulder, N.T.Quach and, M.J.Flynn, "An Area Model for On-Chip Memories and its Application, "Journal of Solid State Circuits, 26(2), Feb. 1991.

[187] S.Przybylski, M.Horowitx and, J.Hennessy, "Characteristics of Performance Optimal Multi-Level Cache Hierarchies, "Proc. of the 16th Annual Symp. on Comp. Arch., pp.114-121, June 1989.

[188] M.R.Ransford, "MC 68040 Cache Design Study, "NCR Journal, 3(2), Dec., 1989.

[189] M.Squillante and E.Lazouska, "Using Processor-Cache-Affinity Information in Shared-Memory Multiprocessor Scheduling, "IEEE Transactions on Parallel and Distributed Systems, 4(2), pp.131-143,

Feb., 1993.

[190] N.Suzuki, Shared Memory Multiprocessing, "MIT Press, Cambridge, MA, 1992.

[191] M.Thapar, Cache Coherence for Scalable Shared Memory Multiprocessors, Ph.D. Thesis, Stanford University, CSL-TR-92-522, May 1992.

[192] M.Tomasevic and V.Milutinoivc, "A Simulation Study of Snoopy Cache Coherence Protocols, "Proc. of the HICSS, Koloa, Hawaii, pp.427-436, Jan., 1992.

[193] J.Torrellas, A.Tucker and, A.Gupta, "Benefits of Cache-Affinity Scheduling in Shared-Memory Multiprocessor, "Performance Evaluation Review, 21(1), pp.272-274, June, 1993.

[194] M.S.Obaidat and H.Khalid, "Ultrasonic Transducer Characterization By Neural Networks, "Submitted for publication in Neural Computing and Applications Journal, 1996.

[195] M.S.Obaidat and H.Khalid, "Performance Evaluation of Neural Network Paradigms for The Characterization of Ultrasonic Transducers, "Proceedings of the 1995 IEEE Annual International Conference on Electronics, Circuits and, Systems (ICECS'95), pp.370-376, December 17-21, 1995.

[196] H.Travén, "A Neural Network Approach To Statistical Pattern Classification by Semiparametric Estimation of Probability Density Functions, "IEEE Transactions on Neural Networks, 2(3), pp.366-377, May, 1991.

[197] J.Hertz, A.Krogh and, R.G.Palmer, Introduction To The Theory of Neural Computation, Addison-Wesley Publishing Company, Reading, MA, 1991.

[198] J.M. Zurada, Introduction To Artificial Neural Systems, West Publishing Company, 1992.

[199] M.S.Obaidat and D.S.Abu-Saymeh, "Methodologies for Characterizing Ultrasonic Transducers Using Neural Network and Pattern Recognition Techniques, "IEEE Trans. on Industrial Electronics, 39(6), pp.529-536,

December, 1992.

[200] B.Widrow and M.A. Lehr, "30 Years of Adaptive Neural Networks: Perceptron, Madaline and, Backpropagation, "Proc. IEEE, 78(9), pp.1415-1442, September, 1990.

[201] W.Lin, F.Liao, C.Tsao and, T.Lingutla, "A Hierarchical Multiple-View Approach to Three-Dimensional Object Recognition, "IEEE Trans. on Neural Networks, 2(1), pp.84-92, January, 1991.

[202] K.Warwick, G.W.Irwin and, K.J.Hunt, Neural Networks for Control and Systems, Published by Peter Peregrinus Ltd., U.K., IEE Control Engineering Series 16, 1992.

[203] S.C.Ahalt, A.K.Krishnamurthy, P.Chen and, D.E.Melton, "Competitive learning Algorithms for Vector Quantization, "Neural Networks, 3, pp.277-290, 1990.

[204] L.B.Almeida, "Backpropagation in Perceptrons with Feedback, "Neural Computers, Berlin:Springer-Verlag, pp.199-208, 1988.

[205] J.A.Anderson and E.Rosenfeld, Neurocomputing: Foundations of Research, Cambridge, MIT Press, 1988.

[206] P.Baldi and K.Hornik, "Neural Networks and Principal Component Analysis:Learning from Examples Without Local Minima, "Neural Networks, 2, pp.53-58, 1989.

[207] A.G.Barto and P.Anandan, "Pattern Recognizing Stochastic learning Automata, "IEEE Transactions on Systems, Man and, Cybernetics, 15, pp.360-375, 1985.

[208] A.G.barto and M.I.Jordan, "Gradient Following Without Backpropagation in Layered Networks, "IEEE First International Conference on Neural Networks, San Diego, pp.629-636, 1987.

[209] S.Becker and Y.Le Cun, "Improving the Convergence of Backpropagation learning with Second Order Methods, "Proc. of the 1988 Connectionist Models

Summer School, Pittsburg, pp.29-37, 1989.

[210] H.D.Block, "The Perceptron: A Model for Brain Functioning, "Reviews of Modern Physics, 34, pp.123-135, 1988.

[211] A.Blumer, A.Ehrenfeucht, D.Haussler and, M.Warmuth, "Classifying Learnable Geometric Concepts withthe Vapnik-Chervonenkis Dimension, "Proc. of the 18th Annual ACM Symp. on the Theory of Computing, Berkeley, pp.273-282, 1986.

[212] S.Brunak and B.Lautrup, Neural Networks:Computers with Intuition, World Scientific Publisher, Singapore, 1990.

[213] Y.Chauvin, A Backpropagation Algorithm with Optimal Use of Hidden Units, Advances in Neural Information Processing Systems I, Morgan Kaufmann Publisher, pp.519-526, 1989.

[214] S.Diederich and M.Opper, "Learning of Correlated Patterns in Spin-Glass Networks by Local Learning Rules, "Physical Review Letters, 58, pp.949-952, 1987.

[215] S.E.Fahlman, "Fast-learning Variations on Backpropagation: An Empirical Study, "Proc. of the 1988 Connectionist Models Summer School, Pittsburg, pp.38-51, 1989.

[216] D.Farmer and J.Sidorowich, Expoliting Chaos to Predict the Future and Redure Noise, In Evolution, Learning and Cognition, World Scientific Publisher, Singapore, pp. 277-330, 1988.

[217] D.E.Goldberg, Genetic Algorithms in Search, Optimization and, Machine Learning, Addison-Wesley, Reading, MA, 1989.

[218] R.P.Gorman and T.J.Sejnowski, "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Target, "Neural Networks, 1, pp.75-89, 1988.

[219] D.H.Graf and W.R.LaLonde, "A Neural Controller for Controller for Collision-Free Movement of General Robot Manipulators, "IEEE International Conference on Neural Networks, San Diego, 1, pp.77-84, 1988.

[220] S.Grossberg, "Neural Expectation: Cerebellar and Retinal Analogs of Cells Fired by Learnable or Unlearned Pattern Classes, "Kybernetik, 10, pp.49-57, 1972.

[221] G.Gyorgyi, "Inference of a Rule by a Neural Network with Thermal Noise, "Physical Review Letters, 64, pp.2957-2960, 1990.

[222] G.Gyorgyi and N.Tishby, Statistical Theory of Learning a Rule, Neural Networks and Spin Glasses, World Scientific Publishers, 1990.

[223] S.J.Hanson and L.Pratt, A Comparison of Different Biases for Minimal Network Construction with Backpropagation, Advances in Neural Information Processing Systems I, Morgan Kaufmann, pp.177-185, 1989

[224] S.A.Harp, T.Samad and A.Guha, Designing Application-Specific Neural networks Using the Genetic Algorithm, Advances in Neural Informatin Processing Sustems II, Morgan Kaufmann Publishers, pp.447-454, 1990.

[225] E.J.Hartman, J.D.keeler and J.M.Kowalski, "Layered neural networks with Gaussian Hidden Units As Universal Approximations, "Neural Computation, 2, 210-215, 1990.

[226] R. H.-Hielsen, "Theory of the Backpropagation Neural Network, "IEEE International Joint Conference on Neural Networks, Washington, 1, pp.593-605, 1989.

[227] J.A.Hertz, Statistical Dynamics of Learning, Preprint 90/34 S, Nordita, Copenhagen, Denmark.

[228] G.E.Hinton, "Learning Distributed Representations of Concepts, "Proc. of the 18th Annual Conference of the Cognitive Science Society, Amherst, pp.1-12, 1986.

[229] J.J.Hopfield, "Learning Algorithms and Probability Distributions in Feed-Forward and Feed-Back Networks, "Proc. of the National Academy of Sciences, USA, 84, pp.8429-8433, 1987.

[230] J.J.Hopfield, D.I.Feinstein and, R.G.Palmer, "Unlearning Has a Stabilizing Effect in Collective Memories, "Nature, 304, pp.158-159, 1983.

[231] J.J.Hopfield and D.W.Tank, Neural Architecture and Biophysics for Sequence Recognition, Neural Models of Plasticity, Academic Press Publishers, 1989.

[232] K.Hornik, M.Stinchcombe and H.White, "Multilayer Feedforward Networks Are Universal Approximators, "Neural Networks, 2, 359-366, 1989.

[233] K.Hsu, D.Brady and, D.Psaltis, "Experimental Demonstration of Optical Neural Computers, "Neural Information Processing Systems, American Institute of Physics, pp.377-386, 1988.

[234] D.R.Hush and J.M.Salas, "Improving the Learning Rate of Backpropagation with the Gradient Reuse Algorithm, "IEEE International Conference on Neural Networks, San Diego, 1, pp.441-447, 1988.

[235] R.A.Jacobs, "Increased Rates of Convergence Through Learning Rate Adaptation, "Neural Networks, 1,

pp.295-307, 1988.

[236] W.Kinzel and M.Opper, Dynamics of Learning, Physics of Neural Networks:Springer-Verlag, 1, 1990.

[237] T.Kohonen, G.Barna and, R.Chrisley, "Statistical Pattern Recognition with Neural Networks: Benchmarking Studies, "IEEE International Conference on Neural Networks, San Diego, 1, pp.61-68, 1988.

[238] A.H.Kramer and A. Sangiovanni-Vincentelli, Efficient Parallel Learning Algorithms for Neural Networks, Advances in Neural Information Processing Systems I, Morgan Kaufmann Publishers, pp.40-48, 1989.

[239] Y.LeCun, B.Boser, J.S.Denker, D.Henderson, R.E.Howard, W.Hubbard and, L.D.Jackel, Handwritten Digit Recogniton with Backpropagation Network, Advances in Neural Information Processing Systems II, Morgan Kaufmann Publishers, pp.396-404, 1990.

[240] S.M.-Ebeid, J.-A.Sirat and, J.-R.Viala, "A Rationalized Backpropagation Learning Algorithm, "IEEE International Joint Conference on Neural Networks,

Vol.II, Washington, pp.373-380, 1989.

[241] C.Mead, Analog VLSI and Neural Systems, Addison-Wesley, Reading, MA, 1989.

[242] M.Mezard and J.-P. Nadal, "Learning in Feedforward Layered Networks: The Tiling Algorithm, "Journal of Physics, A 22, pp.2191-2204, 1989.

[243] G.F.Miller, P.M.Todd and, S.U.Hegede, "Designing Neural Networks Using Genetic Algorithms, "Proc. of the 3rd International Conference on Genetic Algorithms, San Mateo, pp.379-384, 1989.

[244] N.M.Nasrabadi and Y.Feng, "Vector Quantization of Images Based upon the Kohonen Self-Organizing Feature Maps, "IEEE Internatinal Conference on Neural Networks, San Diego, 1, pp.101-108, 1988.

[245] J.Taylor and K.P.Li, "Analysis of a Neural Network Algorithm for Vector Quantization of Speech Parameters, "Neural Networks Supplement, 1, pp.310- , 1988.

[246] T.Poggio and F.Girosi, "Regularization Algorithms for Learning That Are Equivalent to Multilayer Networks, "Science, 247, pp.978-982, 1990.

[247] U.Riedel, R.Kuhn and J.L.vanHemmen, "Temporal Sequences and Chaos in Neural Nets, "Physical Review, A 38, pp.1105-1108, 1988.

[248] D.F.Specht, "A General Regression neural Network, "IEEE Trans. on Neural Networks, 2(6), pp.568-576, 1991.

[249] R.A.Jacobs, M.I.Jordan, S.J.Nowlan and G.E.Hinton, "Adaptive Mixtures of Local Experts, "Neural Computation, 3, pp.79-87, 1991.

[250] M.I.Jordan and R.A.Jacobs, "Hierarchies of Adaptive Experts, "Advances in Neural Information Processing Systems, 4, pp.985-992, 1992.

[251] J.A.Leonard and M.A.Kramer, "Radial Basis Functions for Classifying Process Faults, "IEEE Control Systems, pp.31-38, April, 1991.

[252] D.F.Specht, "Probabilistic Neural Networks, "Neural Networks, Nov., 1990.

[253] J.Voldman et al., "Fractal Nature of Software-Cache Interaction, "IBM Journal of Research and Development, 27(2), pp.164-170, March, 1983.

[254] SPEC Newsletter, A Summary of the SPEC Benchmark Suites, 6(1), pp.4, March, 1994.